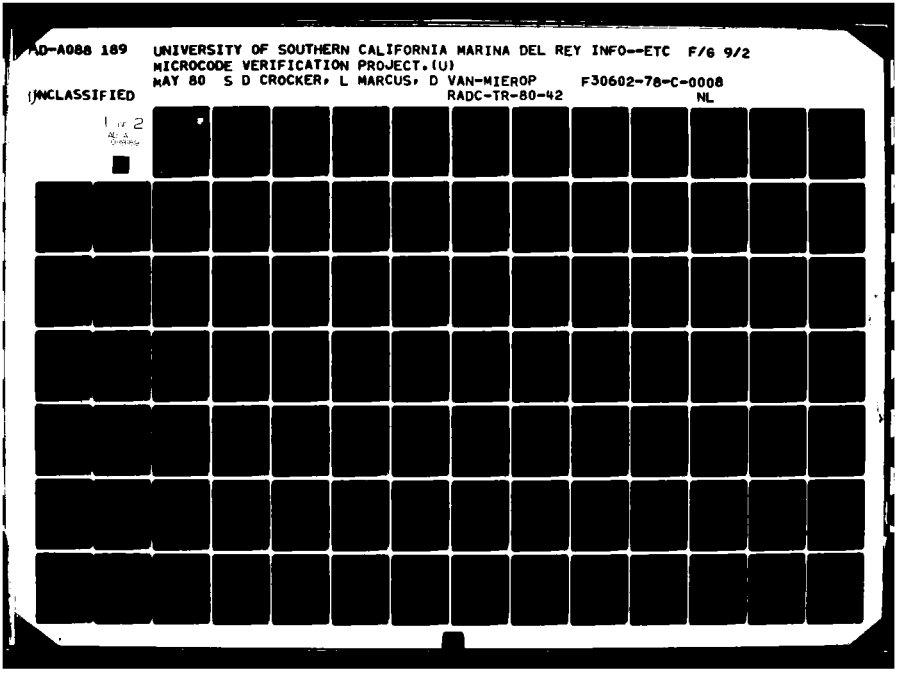
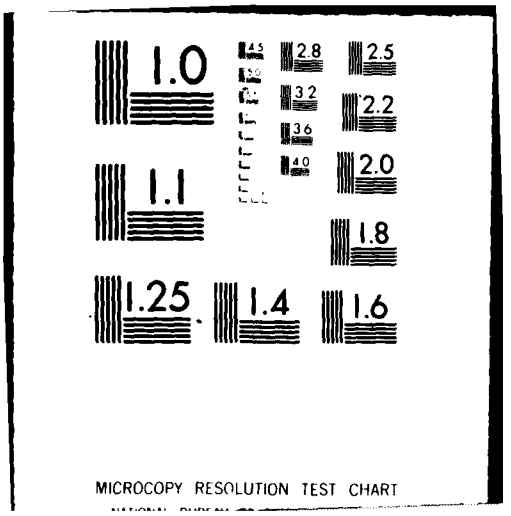
AD-A088 189 UNIVERSITY OF SOUTHERN CALIFORNIA MARINA DEL REY INFO--ETC F/G 9/2
MICROCODE VERIFICATION PROJECT.(U)
MAY 80 S D CROCKER, L MARCUS, D VAN-MIEROP F30602-78-C-0008
UNCLASSIFIED RADC-TR-80-42 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART

LEVEL II

AD A082 461

(12)

RADC-TR-80-42
Final Technical Report
May 1980

MICROCODE VERIFICATION PROJECT

University of Southern California

Stephen D. Crocker
Leo Marcus
Dono van-Mierop

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-80-42 has been reviewed and is approved for publication.

APPROVED: Donald F. Roberts

DONALD F. ROBERTS
Project Engineer

APPROVED: Wendall C. Bauman

WENDALL C. BAUMAN, Colonel, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISIS) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(19) REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER (18) RADC-TR-80-42	2. GOVT ACCESSION NO. AD-A088189	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) MICROCODE VERIFICATION PROJECT.	5. TYPE OF REPORT & PERIOD COVERED (1) Final Technical Report.
	6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s) (10) Stephen D. Crocker, Leo Marcus, Dono van-Mierop	8. CONTRACT OR GRANT NUMBER(s) (15) F30602-78-C-0008
9. PERFORMING ORGANIZATION NAME AND ADDRESS University of Southern California Information Sciences Institute, 4676 Admiralty Way, Marina del Rey CA 90291	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62702F (16) 55812007 (17) 20
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISIS) Griffiss AFB NY 13441 (11)	12. REPORT DATE May 80
	13. NUMBER OF PAGES 144
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Same (12) 149	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Donald F. Roberts (ISIS)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

ISPS	proof checker
microcode	simplifier
program verification	state deltas
symbolic simulation	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The goal of the microcode verification project at ISI is the development of both theory and tools for verification of microcode. Within the scope of this project, a formalism for representing state transitions in a computationally tractable way has been invented, and a proof system based on this formalism has been designed and implemented. The representations of state transitions are called "state deltas."

(Cont'd)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

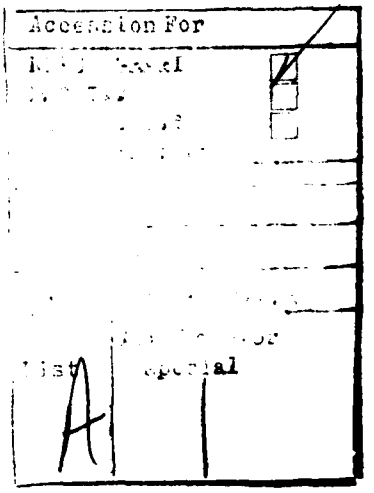
40715

The basic proof system has been specialized for proofs about machine language and microcode by the addition of simplification rules for bitstring arithmetic, and by the addition of a translator from the ISPS machine description language to state deltas.

Some experimentation with the system has been driven by a preliminary attempt to verify parts of the microcode of the Fault-Tolerant Spaceborne Computer (FTSC). The primary success to date has been the verification of the basic algorithm used for computing floating point square root.

Table of Contents

List of Figures



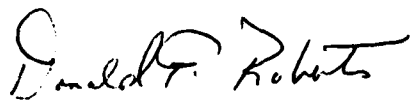
EVALUATION

One goal of Software Engineering Tools and Methods, a subthrust of TPO-5 Software Cost Reduction, is the development of automated tools for use in the production, testing, and maintenance of Air Force software. This effort was undertaken in response to that goal.

The objective of the effort was to develop a prototype software system for formally verifying microcode. The use of microcode (firmware) to implement computer instruction sets, rather than hard wiring, is a recent development in computer technology. Hardware diagnostics do not fulfill testing requirements for these computers.

Formal proof-of-correctness techniques, previously developed, were applied to develop a system for "proving" microcode correctness. These techniques were developed for software written in high order languages. This effort is significant in that it is the first application of the techniques on assembly or micro level software.

Development of the system was guided by problems encountered in attempting to verify the microcoded instruction set of the SAMSO Fault Tolerant Space Computer (FTSC). This provided a practical problem to demonstrate the usefulness of the system. Verification of the complete FTSC instruction set will be completed in a follow-on effort sponsored by SAMSO.



DONALD F. ROBERTS
Project Engineer

1. OVERVIEW

The goal of the microcode verification project at ISI is to develop both the theory and the tools for verification of microcode. While some prior work has been done in this area, notably [Patterson 77, Birman & Joyner 76], the field was (and is) far from closed. Problems exist at every level, from fundamental questions of theory through questions of strategies of system design to problems of integration with other software engineering tools and education of users. Our strategy has been to concentrate on developing a working system, letting the theoretical issues emerge--sometimes painfully--amid system development. We have tried to delay overall consideration of the human engineering questions, but have been forced to consider some of these when it became too difficult to use our own system without improving the interface.

To establish a focus for the project and provide a source of examples, we selected a particular computer, the Fault-Tolerant Spaceborne Computer (FTSC), under development by Raytheon for the Space and Missile Systems Organization (SAMSO) of the Air Force. The FTSC has a number of unusual features related to its design goal for a five-year maintenance-free survival in space. These features appear primarily at the hardware level and in the operating system, however, not in the architecture seen or implemented by the microcode. At the machine language level, the programmer sees a 32-bit machine with 64K of memory, 8 general purpose registers and the usual types of instructions. At the microcode level, the machine is horizontally microprogrammed with 78-bit instructions decoded into 37 different fields. (As of this writing, the machine has been redesigned to have a shorter microinstruction. We have not taken these changes into account in the present work, but will focus on the new design in the next effort.) Documentation of the FTSC is given in [Raytheon Corp 76].

One of the criteria in the selection of the FTSC is that it is a real machine developed outside our control. We believe that it is possible to verify code for nearly arbitrary machines, irrespective of the techniques used to develop the code. This view differs somewhat from those of other verification researchers, notably [London 77]. To be fair, it is quite clear that much of the labor in the verification task can be reduced if verification and code development are carried out together and if the strategies,

practices, and tools used to develop the code are also geared toward verification. But we view this as a secondary concern and not fundamental to the verification task. Below, we will mention where the savings in labor would occur.

We view a microprogram verification system in the following terms. A user prepares formal descriptions of the ***host machine*** and the ***target instruction set***. He also obtains a copy of the microcode that runs on the host machine and allegedly implements the target instruction set. He then prepares a proof that the microcode does indeed behave as desired, and submits all four of these files--host description, microcode, target description, proof--to the verification system, which then examines the target description to determine all aspects of its behavior needing implementation. For each sequence of events that must be implemented, the system symbolically executes the microcode according to the rules of the host machine and demonstrates that the required sequence of events does take place.

No system can be quite smart enough to carry out all possible demonstrations completely automatically, so some help may be needed. Some systems operate on the principle that the system should try very hard to succeed on its own and then ask for help after it has tried all possible heuristics. While this approach seems attractive, it has a fundamental drawback. When the system asks the user for help, the user is generally unaware of what the system already has tried to do, what level of detail is needed, or even what problem the system is working on. The underlying difficulty is that the user must have some idea of how the system is constructed and understand how to drive the system. At the same time, we note that the system is really trying to formally document the rationale for each instruction in the microprogram. However, this is just what the programmer had to do himself when he wrote the program. Combining these two observations, we have taken the view that the verification system should be driven by the user, not the other way around. The user should have a complete understanding of what the verification system will and will not do, and the user should drive the verification system toward believing the correctness of the code. In this view, interaction between the system and the user takes the form of a prepared proof, and it becomes meaningful to ask what is the proper language for writing proofs. Wegbreit's

paper [Wegbreit 77] explores this area elegantly for well-structured algorithmic languages. For microcode generated with minimal assembly language tools, different engineering is required, but the basic idea is the same. At the present time, our "proof language" is nothing more than a set of commands to the proofchecker. However, as we gain experience with the system, it becomes clear how to structure these commands into phrases; thus the development of a proof language begins. At the same time, it is worthwhile to ask whether the production of both the microcode and the proof of its correctness can share any tools. The answer must be "yes," but we have not yet considered any specific implementation.

Although we wish our system to be as general and as useful as possible, our present design horizons embody the following limitations:

- The purpose of the microcode must be to implement the instruction set of a computer. This restriction is intended to limit the difficulty of specifying the intended behavior of the microcode. With this restriction, we rule out microcode that is just arbitrary lower level code to implement, say, operating systems, signal processing algorithms, device controllers, etc. This restriction is not really fundamental to our work and, as we shall see, does not quite guarantee that we shall always have a straightforward way to specify the intended behavior of the machine.

- Since we do not yet have sufficient tools to represent or reason about concurrency or time-dependent behavior, we demand that our microcode be written for a sequential machine and that it implement the instruction set of a sequential machine.

- We intend that the result of this research be a demonstrable system with the real possibility that someone other than ourselves should be able to formulate a task and carry it out. We do not intend, however, that the system be efficient, completely robust, smoothly human-engineered, or thoroughly documented. Users of the system should understand the state of development. Their success rate will be higher if they communicate with us before and during any experimentation.

In addition to the caveats above, the system we are building is not yet ready for release.

Carrying out a complete proof may be fairly tedious. Preparation of the formal

descriptions often appears to be a straighforward task of encoding the information in the manuals that accompany the machine, but we have noticed that many important details are often omitted from such documents, and others are misdocumented. Programmers developing the microcode come to understand these details and use their knowledge to write or debug their code. If the person writing the formal description is not similarly steeped in the culture of the machine under consideration, a similar learning period will be required.

Writing the proof may be tedious, for three reasons. First, a complete understanding of the code is necessary. The programmer understands the code; the person responsible for verification may not. A period of study may be necessary before any of the proof can be written. Of course, if the programmer were also responsible for preparation of the proof, then the verification would proceed all the faster. Unfortunately, with verification still in the research phase, programmers who build "real" programs are far too busy to spend the extra time required for verification. Also, since verification requires some special knowledge, production programmers may not be skilled in the art of preparing formal descriptions and proofs.

The second difficulty is that the code may be relatively complicated to verify. At the beginning we insisted that it should be possible to verify code even if it were written without knowledge that it would be subjected to verification. (We're assuming, of course, that the code does indeed work!) However, it is equally clear that there are many strategies for writing code and that some of them may be equally good from the programmer's point of view but require very different levels of effort in verification.

The third difficulty is that proofs may be tediously long. We have said that the user must drive the verification system with a proof and that the verification system must proceed so as to give the user a clear idea of what the system is doing. However, a trivial way to build such a system is to make it extremely simple, with the result that proofs will be extremely long and require the user to spend a long time preparing them. In the extreme, this is not permissible; it is necessary to build the system with enough knowledge so the "straightforward" deductions are carried out automatically. There is no possibility that any system can know a "maximum" of knowledge, for there will always

be problems that can be proven with a system, but not proven automatically. At the same time, there is no limit to making a system smarter; we can always go beyond the previous limits and build a next system that understands more than the last. Clear measures of the smartness of one system compared to another do not yet exist, but it is a question that is likely to gain attention as various verification systems are used for larger and larger problems.

As we said earlier, we have restricted our interest to microcode that implements the instruction set of some computer. The intention of this limitation is to make it easy to specify the intended behavior. Unfortunately, this restriction does not quite work. In the description of the host architecture, we have no difficulty in formalizing all aspects of concern, excepting, of course, timing and concurrency. We view the host machine as operating on bitstrings of finite length. The operators for bitstrings are concatenation and selection, logical operations, e.g., AND, OR and NOT, and the simple integer arithmetic operations. At the target level, however, we have not been so fortunate. Bitstrings remain the dominant datatype, and all of the bitstring operators are still required, but new operations exist that are not simply characterized by short descriptions. Floating point arithmetic is the most obvious and extensive area, but some machines have other instructions whose behavior is quite difficult to characterize in terms of bitstrings. Edit and format instructions provide many examples, as do instructions that find the lowest-order or higher-order 1 bit.

The FTSC computer is blessed with the usual complement of floating point instructions; indeed, it even has a floating point square root instruction. On the grounds that avoiding these instructions would trivialize the effort and leave us an undetermined distance from realizing a system capable of verifying real microprograms for real machines, we decided to tackle the floating point arithmetic head on.

We divided the specification of the target machine into two levels. The first is written in the same terms as the host machine description. It is restricted to simple bitstring operators. At this level, the simple target machine instructions, e.g., load, store, integer add, jump, etc., are stated as succinctly as they will ever be stated and no further work is required. The floating point instructions, however, look like short but complicated

algorithms that provide an explicit view of how the words are divided into a mantissa and exponent, how normalization takes place, etc.

For these instructions, we provide a higher level of specification that shows that the result of that algorithmic specification has certain properties. This higher level of specification requires the introduction of the reals, and the properties are stated in terms of the interpretation of the floating point bitstrings as real numbers. For example, the desired property of the square root instruction is that it computes the largest floating point number whose square is not larger than the original number. (The notion of "largest floating point number" requires even a little more; the granularity of the floating point numbers is also an issue.)

In the work to date, we have written a complete specification of the FTSC at both the host and algorithmic target level, but we have not defined the properties required of the floating point instructions except for the square root instruction. We have focused on the square root instruction simply because it seemed to expose all of the issues likely to come up in any other instruction.

The basic plan for verifying the correctness of the microcode thus has two parts. One part is to verify that the microcode running on the host machine implements the algorithmic target level. The second part is to verify that the algorithmic target level has the additional properities desired.

At the present time, we have completed the proof that the algorithmic target description of the square root instruction has the desired property. We have not yet proven similar properties for other instructions, nor have we proven the correspondence between the host machine and the target instruction set, *for the FTSC*. We have, however, created a simple, fictitious machine and carried out a complete proof of the correctness of its microcode. This small machine is called the TOY machine. Both of these proofs are documented in chapter four.

Completion of proofs is one measure of progress, but there is much that precedes the ability to carry out proofs. A sound theoretical basis must exist or be developed and a functioning proof system must be developed. These activities have consumed the

majority of our time and resources.

In chapter two, we discuss the theoretical basis for our proof system and introduce the language we use for expressing the behavior of machines and the properties of programs. In chapter three, we outline the structure of the proof system and give details for selected components.

This work is still in progress. The details of language, structure and capabilities are all evolving.

2. LANGUAGE AND THEORY

In this chapter we discuss the formal basis of and the language we have chosen for both encoding our descriptions of machines and reasoning about the course of computations. Internally, our notation is chosen for its precision and ease of processing, qualities that contrast with the desire for compactness and richness in the languages read and written by humans. Both levels exist, and there must be translation between them. As often happens, subtle and important issues emerge in the translation. At IBM, the difficulties of using two levels of language have been avoided by designing a special-purpose language that is both computationally tractable and not too unwieldy for humans. That language is documented in [Joyner et al. 78].

2.1 ISPS

To represent the host and target machines, we have chosen to use the ISPS language. ISPS, a derivative of Bell and Newell's ISP language [Bell and Newell 71], is now in modest use by a number of organizations. Documentation of the current version is given in [Barbacci et al. 77]; the examples in chapter four are written in ISPS.

Descriptions of machines have been written in ISPS for a number of different purposes, including simulation, architecture evaluation, documentation, computer-aided design, and (in variants of ISPS) automatic generation of code generators and assemblers. This variety of activity associated with the language is useful in two ways. On the one hand, the use by large numbers of people improves the possibility that a standard will emerge, that documentation of computers will be more accurate and more complete, and that the task of preparing formal descriptions of the host and target levels of a microprogrammed machine will be carried out by the machine designers instead of by the verification group.

On the other hand, the wide variety of applications using ISPS, each with its own software to process ISPS descriptions, has tended to expose the lack of a precise semantics for the language. As an experiment to gain some leverage on the semantics of ISPS, Pete Alfvin developed a denotational semantic definition of AMDL, an abstract syntax version of ISPS in use at ISI [Alfvin 79].

As we mentioned in the overview, while it may look simple to encode the details of a machine's instruction set in ISPS, it may be tedious in actuality. In the case of the FTSC, a machine under development and redesign, a number of small but important details were either undocumented or misdocumented. We developed simulation tools to execute the descriptions we wrote and used the simulations to execute the diagnostics for the machine at both the host and target levels. In essence, this amounted to a "verification by testing" approach; since the microcode itself was used in some of these tests, it is reasonable to ask if we perturbed the description of the machine in order to make the code work. Stated another way, how do we know that the description of the host machine is an accurate representation of how the hardware really works, and how do we know that the description of the target machine is an accurate representation of how the target machine is supposed to work? There can be no completely satisfactory answers to these questions. The descriptions at both levels must be accepted; they cannot be checked in any rigorous sense within the confines of the microcode verification paradigm. If there exists another description at a higher or lower level, then the corresponding descriptions may be checked against it. However, this merely pushes the problem off one level, and there is no ultimate exemption from a requirement to accept the bottom level description as the way the machine actually works and the top level description as the way the system is supposed to work.

Complete assurance having been denied us, we can ask what lesser assurance is available. By using a language understood by a number of people (in particular by the designers of the machine, the microprogrammers of the machine, and the programmers at the assembly language level) we can have some hope that they all share the same understanding of the machine if they were to depend upon the same descriptions as their reference. This is not yet the case for any machine with any description system, but we see no reason why it could not be. In the course of writing the formal descriptions, the "outsider" may find himself in a question and answer dialogue with the machine designers, in order to clarify the informal descriptions. See the appendices for an example of our dialogue with the designers of the FTSC.

To complete our discussion of ISPS, we again mention that ISPS does not provide

primitives for representing floating point operations; we have had to code them in ISPS as small algorithms. Since the lack of standard notions and designs of floating point arithmetic is a common problem, the choice of another language would not have improved matters.

2.2 STATE DELTAS

In order to build a proof system, a formal basis for reasoning about machines is required. Ordinary first-order predicate calculus is often used as a foundation, but it provides no machinery for reasoning about time or situations that change with time.

There are many possible solutions. Ours has been the development of an extension to the first-order predicate calculus by the addition of sentences called ***state deltas***. State deltas were first introduced in [Crocker 77]. For a more formal treatment see [Marcus 79]. To motivate the development of state deltas, we give the observations and decisions that support our formulation.

- *It is simple to think in theoretical terms* that a computer can be characterized by a transition function that maps state vectors into state vectors. Given an initial state vector and a statement of the transition function, ordinary mathematical tools will provide the machinery for reasoning about successive states of the machine. However, direct use of this approach becomes unwieldy for even the simplest example.

- One of the first difficulties is the description of the state vector. It is quite inconvenient to think of the state vector as a single domain. For all real machines, the state vector is a messy patchwork of various domains. Each of the storage locations in the machine is a piece of the state vector. The primary memory is perhaps the most regular component, but there are many other components. Also, it may be desirable to subdivide the memory into smaller pieces. To deal with this, we use the usual programming practice of assigning names to different ***places***. A place is essentially a component of the state vector. Given the list of places that comprise the state vector, we will not actually need to symbolize the state vector as a single object. We will not even need to know exactly how the components comprise the state vector, e.g., it is not necessary to know if the state vector is represented as a tuple or whether the program counter is, say, the first or second element of that tuple.

- The precise granularity of time is not really of interest. We do not care

whether a particular computation takes one or two time steps. Instead, we care that certain states follow one another eventually. Accordingly, we avoid describing individual transitions and describe the effect of multiple transitions instead. The result is quite similar to Manna and Waldinger's ***intermittent assertion*** idea [Manna & Waldinger 78], which is derived from Burstall's paper [Burstall 74]. We make use of a ***precondition*** and a ***postcondition***, and our state delta encodes the idea that

> if the precondition holds at some point in time,
>
> then there will be a later time at which the postcondition holds.

- While it might be possible to state the behavior of a machine in a single sentence, it would be quite unwieldy. We make use of a collection of state deltas to specify the behavior of a machine. Each state delta defines the behavior of the machine in only particular circumstances. Of course, it is not necessary to cover all possible circumstances; it is perfectly reasonable to leave the behavior of the machine undefined in some cases.

- Most of the components of the state vector are unchanged at each step. Any straightforward description of the transition function would be dominated by simple statements of equality between large sections of the old and new states. To reduce this burden, our formalism encodes the assumption that all of the state remains unchanged except for a list of places in the state vector explicitly named. Accordingly, a state delta has a ***modification list***. The semantics of a state delta includes

> if the precondition holds at some point in time,
>
> then there will come a time at which the new state is the same as the present state except possibly for the values in the places listed in the modification list, and
>
> at that time the postcondition will also hold.

- Even with the implicit assumption that most of the state remains unchanged from one state to another, it may be necessary to include many details in the precondition. Quite often the precondition includes the requirement that much of the present state is identical to a particular prior state. This introduces a third time into the formalism. We have encoded this condition with another list of places, called the ***environment list***. The semantics of state delta are now stated as

> if the contents of the places listed in the environment list are the same at some time t_1 as they were at an earlier time t_0, and

if the precondition is true at time t_1,

then there will be a later time t_2 in which the new state is the same as the state at time t_1 *everywhere* except possibly at the places listed in the modification list, and

the postcondition will also hold.

- To simplify our bookkeeping about times and states, we organize all of our thoughts in terms of a current time. In the formulation above, we anchor t_0 to the current time. We can restate the formulation as

  if at some future time t_1 all of the values in the places listed in the environment list are the same as they are now, and

  if the precondition holds at that time,

  then there will come a time t_2 whose values are the same as at time t_1 everywhere except possibly in the places list in the modification list, and

  the postcondition will hold.

- While this formulation is quite close to what we need to support efficient reasoning about places and states, the requirements imposed by the *modification and environment lists are more difficult* than they look. As stated, it is permitted that the values inside the environment list and outside the modification may change in the interim, as long as they are restored at the end of the interval. We have found it more useful to tighten this requirement so that the values that must be the same at the ends of the time intervals are in fact never changed during the intervals. It turns out that tightening the restriction of the environment and modification lists does not remove any essential power. On the contrary, this new version allows the restricted use of the modal operator "during" to form sentences which are not expressible using only pre- and postconditions. Our formulation is now

  if the values listed in the environment list remain unchanged from now until some future time, and

  if the precondition also holds at that time,

  then at the end of some succeeding time interval during which at most only the values listed in the modification list will have changed, and

the postcondition will hold.

Note that there is no requirement that values that are unchanged from now until the precondition becomes true remain unchanged when the postcondition becomes true. In other words, it is possible that the same place may be listed in both the environment and modification lists. Later, we will see the use and effect of such an intersection.

The syntactical form of a state delta is

```
(SD     (pre: P)
        (mod: M)
        (env: E)
        (post: Q))
```

where P and Q are usually first order sentences in some language, but may in fact be state deltas themselves, and M is a list of places, as is E. See Chapter 4 for additional examples of state deltas.

Note that the logical implication P implies Q (in a given state) is equivalent to the state delta

```
(SD     (pre: P)
        (mod: )
        (env: OMEGA)
        (post: Q))
```

being true in that state, where OMEGA is a list of all places, or equivalently a single state "containing" all others.

Also note that one state delta may be derived from two others by a kind of case analysis.

If

```
(SD     (pre: P AND P')
        (mod: M)
        (env: E)
        (post: Q))
```

and

```
(SD     (pre: P AND (NOT P'))
        (mod: M)
        (env: E)
        (post: Q))
```

hold in a certain state, then

```
(SD     (pre: P)
        (mod: M)
        (env: E)
        (post: Q))
```

holds in that state.

An important tool is the "dot" operator .R, which when applied to a place R (for "Register") represents the value or contents of that place. Thus a state change entails a redefinition of dot, not a reinterpretation of the place itself.

When dot is used in a state delta it always refers to the contents at the time of the precondition. In order to reference the contents of a place at the time of the postcondition, the symbol # is used. For example,

```
(SD     (pre: .R GTR 0)
        (mod: R)
        (env: )
        (post: #R=.R-1))
```

means that if the value of R is greater than 0, then at some later time the new value will be one less (and nothing changed along the way except for R).

Here is an example of deriving one state delta from another by a form of induction: Assume the contents of places are nonnegative integers. If

```
(SD     (pre: P(.R) AND .R GTR 0)
        (mod: M)
        (env: E)
        (post: P(#R) AND .R GTR #R))
```

holds in a certain state, and in addition if M and E represent disjoint sets of places, then

```
(SD     (pre: P(.R) AND .R GTR 0)
        (mod: M)
        (env: E)
        (post: P(0))
```

holds in that state.

It is obvious how an input-output specification can be stated using state deltas. In the next sections we shall explain how a simulation relation between two programs can be proved using state deltas.

For now let us point out how a set of state deltas can be viewed as a program. Assume that we are given a set of state deltas, ordered in some way, and an "initial" state. The first state delta (according to the above ordering) whose precondition is true in the current state may be "applied", thus transforming the state into that specified by the postcondition (and the modification list). Actually the term "state" should perhaps be replaced by "set of states" since we do not demand that the postcondition completely determine the state; for example, the actual values of some places may not be determined, but rather some properties of these values are known. The components (sentences) of the old state which were dependent on, or "supported by", places in the modification list are removed from the state, and the list of sentences in the postcondition are added to the remaining sentences.

Now the process is repeated in the new state. This process is called symbolic execution.

It is also possible to view a somewhat arbitrary program as a set of state deltas, or to translate a program into state deltas, as is discussed in Section 2.4.

2.3 SIMULATION

As stated in the overview, the process of microcode verification can be divided into two parts: the first showing that the Host Machine implements the Target Machine, the second showing that the Target Machine satisfies the Top Level Specification. We shall now discuss the first of these parts.

Let us think on the level of abstraction where both the host and microcode and the target may be considered as programs A_1, A_2. Intuitively, A_1 simulates A_2 if A_1 can "do" anything A_2 can; that is, the state changes due to A_2 are reflected in the state changes that A_1 causes. The state changes for A_1 and A_2 separately are computed using the symbolic execution of the previous section. To prove that A_1 (symbolically) simulates A_2 we need to establish a correspondence between the states of A_1 and those of A_2 such that given two corresponding states, S_2 (for A_2) and S_1 (for A_1), if S_2' is the next state after S_2 arrived at by executing A_2, then the (a) state S_1' corresponding to S_2' can be arrived at by executing A_1 from S_1 (though S_1' need not be the very next state after S_1).

In the system implementation, a state is specified (as in the precondition or postcondition of a state delta) by a list of first order sentences and SDs, and the correspondence between states is specified by a function called MAPPING. Again, recall that "state" as used here is not necessarily a complete description. Thus MAPPING is actually a correspondence between sets of complete states.

2.4 TRANSLATION OF ISPS INTO SDS

ISPS is a relatively well known language suitable for machine descriptions. We will see that SD notation is suitable for representing intermediate proof steps, performing symbolic execution, and utilizing the efficiency of the modification list. In order to retain the advantage of ISPS as an input language and SDs as an internal notation, we need to translate ISPS descriptions into SDs.

If we invent a place to represent the internal control state of a machine and we assign a symbolic value to the control place for each statement in an ISPS program, the program could be represented with a set of SDs, where each SD represents a possible state change. References to control states could be made by including predicates of the form .PC=label in the precondition and postcondition (PC represents the internal control state "program counter"; "label" represents the control value). Representing all the state changes with SDs has two drawbacks: the thread of control that is implicit in the ISPS representation is lost and is encoded explicitly into the precondition and postcondition; the SD notation is different from the familiar ISPS (and somewhat more complicated).

Nested State Deltas

The scheme we are using is motivated by the need to model the control mechanism inside a machine. In an earlier formulation, we modelled the control mechanism as a single variable that took on explicit values. Each precondition and postcondition mentioned the value, e.g., .MicroPC=A312, and this control place was also mentioned in the modification list of every SD. It did not, of course, occur in the environment list. Since the names of the control state values were completely artificial and the explicit appearance in the pre- and postconditions of these equations was very cumbersome, we revised the

formulation to an entirely equivalent scheme that simply made implicit use of the value of control place. The only property of the control place we cared about is that it made some precondition true. By embedding the next SD in the postcondition of the current SD, the next SD is automatically made valid when the current SD is applied ("executed"). Of course, its validity disappears when the control place is changed, so it is necessary that the name of the control place appear in the environment list of the new SD. This is what gives rise to the appearance of the same control place in both the environment and modification lists. Of course, there are some SDs that will not have the control place in the environment list. The tops of loops need to be around forever, and we must resort to using names for the values of the control place at those points. SDs that exit from blocks will not generally have SDs in their postconditions; instead they will set relevant values of the control place.

Instead of describing a program by *a set of SDs* (one for each possible state change) we could describe it with *one SD* that represents the first state change and has a nested SD that represents the rest of the program in its postcondition. During symbolic execution, the process of applying an SD is repeated. The following happens for each SD application: the appropriate state change is made; the nested SD that represents the rest of the program is added to the current state; and the SD just applied is removed from the current state if it is supported by the (modified) control place.

The TR Notation

The use of the TR notation is a further compression of the translation from ISPS to SDs. We noticed that it was unnecessary to translate an ISPS description entirely into SDs and then work with the SDs. Instead, we embedded the translation process in the operation of the proof system and carried out just one step of the translation at a time. In essence, we now encode the value of the control place as a formula that tells what to do next. That formula is basically ISPS code, with embellishments to tell us where we are in the code and to keep track of the environment established by ISPS scope rules.

To improve the cumbersome notation of nested SDs to represent the tail of a program, we defined a function called TR that maps an ISPS description into an SD or a set of SDs. We distinguish between ISPS descriptions whose first statement is an assignment

statement and those who start with a control change (conditional or unconditional). In case of an assignment, the TR maps an ISPS program into an SD whose precondition is empty; the modlist includes a control place (MicroPC) and the name of the register that is being assigned to; the envlist includes only MicroPC; the postcondition includes the effect of the assignment and a TR whose parameter is the tail of the ISPS program. In case of a control change, the TR maps an ISPS program into a set of SDs. For each SD, the precondition includes the condition that leads to the control change, the modlist and envlist include MicroPC, and the postcondition includes a TR with the corresponding rest of the ISPS program. The symbolic execution using TRs is very similar to nested SDs, except that the rest of the program is represented as a TR applied to an ISPS description.

Marking ISPS Programs

The set of SDs that represents an ISPS program is not unique. We saw that it ranges from an SD for each ISPS statement to a single SD for the whole program. It depends on the "granularity" that the ISPS description is intended to be broken into. This granularity is specified by special markings of the ISPS description: Every SD that is part of the description of a marked ISPS program must cover a path of execution between two markings.

A control state of an ISPS description is a label or a procedure-entry (that specifies the "rest of the program"). A marking is a special kind of control state. The minimum set of markings needed to specify simulation are the entries and exits of all the procedures. Markings could be added in order to allow more SDs (i.e., a finer granularity). They should be added to break all the loops, for simplicity. Marking should also be added in order to avoid covering the same execution path by more than one SD, for efficiency.

The Translation Process

A marking M_i is a "successor" of M_j if M_i belongs to the set of markings that can be reached by symbolic execution from M_j without visiting any other marking. The translation algorithm generates one SD for each path of execution between two succeeding markings that are reachable from the initial one. The number of SDs generated is determined by the granularity (i.e., the number of markings). When showing simulation, we

will usually use a very fine granularity for the lower level machine (the Host) and a coarser one for the Target. The TR function is used for performing the symbolic execution.

For simplicity we will refer in this paragraph to the translation of the target machine. The control place for the target machine is MacroPC.

The following information is accumulated during the symbolic execution for generating each SD: all the "path conditions" that have to be true in order to reach a successor; the list of places that are modified during execution; the new symbolic state. The new SD covers the path of execution between a marking and its successor, and includes the following: in the precondition the accumulated path condition and .MacroPC="initial label"; in the modlist the accumulated modified places and MacroPC; the envlist is empty; in the postcondition the accumulated symbolic state and .MacroPC=label. A concrete example of translation of an ISPS program is shown in a subsequent chapter.

2.5 THE SYSTEM -- OVERVIEW

The system is described in detail in Appendix A. Here we just describe enough to serve as background for the next chapter. For any additional information, see Appendix A.

The MICROVER system consists of the following components: User Interface, ISPS Translator (described in the previous section), Kernel, Data Base, Place System, and Simplifier. The User Interface, with the help of the ISPS Translator, converts the user's input to a sequence of basic proofsteps. The Kernel processes the proofsteps with the help of the Data Base, Place System, and Simplifier. The Data Base keeps track of the current state, the Place System keeps interdependencies among places, and the Simplifier simplifies expressions in the current state.

The Data Base contains facts which may change as the state changes through symbolic execution, say. Thus it contains facts relating to the contents of places (these facts do not necessarily uniquely determine those contents, e.g., contents of A greater than 0), or relating to some arithmetical variables like induction variables.

The Place System holds "permanent" facts about places, for example which places are subplaces of other places. This is the "Covering" relationship:

(Covering A ((B1 L1) ... (Bn Ln)))

means A is a place with disjoint subplaces B1 of length L1, ..., Bn of length Ln.

The MICROVER system as a whole can be thought of as performing deductions involving dynamic statements (state deltas). The Simplifier is the component performing static deductions. Thus the simplifier contains procedures for simplifying expressions in a given state. If the expression is a sentence (e.g., predicate), and the simplified result is T, then that sentence is true in the given state.

3. EXPERIENCE AND EXAMPLES

The bulk of our work has used examples taken from the FTSC. As we outlined in the overview, we have divided the FTSC target description into two levels. One level provides an algorithmic description for the instructions. For the simple instructions, e.g., load, store, and integer arithmetic instructions, this level of description is easy to read and requires no further refinement. However, for the floating point instructions, an algorithmic description of the effect of an instruction is nearly opaque and is useful only to a specialist who needs to track down the detailed results for particular cases. For these instructions, we need to prove that the results guaranteed by the algorithmic description may be understood in terms of some simply stated properties. The square root instruction is the most interesting example in this area, and we have focused most of our attention on proving just the simple property that the effect of the square root instruction as described by the algorithmic description does indeed compute the largest floating point number whose square is not greater than the original number. We felt this example would expose the hardest issues first and provide some chance that the rest of the proof would be comparatively easy. We have not yet determined whether this strategy will be successful.

At the same time, we have been concerned that the mechanics of carrying out a complete proof should be well understood. Accordingly, we have hedged our bets a bit and constructed a very small fictitious example of a microcoded machine, written the microcode to implement a simple instruction set for that machine, and prepared a complete proof. We call the machine the "TOY" machine.

This chapter details the proofs for both of these examples. To give the flavor of a complete proof, we present the TOY machine first.

3.1 THE TOY MACHINE

The TOY machine is a simple microprogrammed machine. We have provided a formal description of its target instruction set and of its host architecture. We have written the microcode for the host level that implements the target instruction set, and we have specified the states in the host and target levels that correspond to each other. Finally,

we have written a set of commands for the proofchecker to guide it toward proving that when the microcode runs on the host machine, it correctly implements the target instruction set. For a problem this simple, the commands to the proofchecker are entirely devoted to setting up the proof. The actual details are carried out completely automatically.

The TARGET Machine

In order to keep this experiment simple, but still deal with a realistic machine, we designed the TARGET machine according to the following requirements:

- 4K-word 16-bit memory
- a 12-bit program counter, a 16-bit accumulator, and a 16-bit instruct register
- infinite indirect addressing
- six possible operations: add, subtract, store, load, skip or negative, jump.

We decided on the following word format:

TOY starts operating by fetching the instruction from location 1 in memory. It proceeds by repeating the cycle of execution and fetching.

Fetching is performed as follows: the machine loads the instruction register from the memory location that the program counter points to; while the indirect bit is set, the 13 least significant bits of the instruction register are overwritten by the contents of the memory location that the effective address (EA) points to; then the program counter is incremented.

The execution performs one of the following operations according to the 3-bit opcode:

add MEM[EA] to the accumulator; subtract MEM[EA] from the accumulator; load the accumulator with MEM[EA]; store the contents of the accumulator in MEM[EA]; skip the next operation if the most significant bit of the accumulator is one (negative accumulate); jump to EA.

The precise ISPS description of the TARGET machine was written according to the English description and is shown in Figure 3-1. The ISPS program is divided into the following declarations: the memory; the registers; the fetching algorithm; the execution algorithm; the main cycle.

The markings we selected in the TARGET machine are the labels MAIN, XFETCH, FLOOP, and EXEC. The paths that the algorithm found were one from MAIN to FETCH, one from FETCH to FLOOP, one from FLOOP to FLOOP, one from FLOOP to EXEC, nine from EXEC to FETCH.

MacroPC is a dummy place that holds the control state (the label) and TInvReg covers the internal registers. The complete set of SDs that the ISPS to SD algorithm found is shown in Figure 3-2. Let us look closer, for example, at the third SD: it describes the path from FLOOP to EXEC which is denoted by .MacroPC=FLOOP in the *pre:* and #MacroPC=EXEC in the *post:*. The *pre:* also includes .IR<12>=0, which is the precondition for taking this particular path. The *post:* includes also the new value of PC, .PC+1.

The HOST Machine and the Microcode

The HOST machine is the actual hardware that implements the TOY machine. Because the goal of this experiment is microprogram verification, we chose a microprogrammed HOST. The HOST machine was somewhat tailored to the TARGET, for simplicity, but still much generality and extendability were maintained. The description of the HOST machine explicates all the details of registers, combination circuits, and data paths.

We decided to keep the microprogram in a 64-word 21-bit ROM. ROM words contain 21-bit microinstructions with the following format:

```
TARGET := BEGIN
        !! Memory !!
        MEM[0:4k]<15:0>

        !! Registers !!
        PC<11:0>,                ! program counter
        ACC<15:0>,               ! accumulator
        IR<15:0>,                ! instruction register
        OPCODE<2:0> := IR<15:13>, ! operation code
        EA<11:0> := IR<11:0>     ! effective address

        !! Instruction.Fetching !!
        XFETCH := BEGIN
                IR ← MEM[PC] NEXT
                FLOOP1 := REPEAT
                        FLOOP := DECODE IR<12> =>
                                        BEGIN
                                        0 := LEAVE FLOOP1,
                                        1 := IR<12:0> ← MEM[EA]
                                        END
                NEXT PC ← PC + 1
                END

        !! Instruction.Execution !!
        EXEC := BEGIN
                DECODE OPCODE =>
                        BEGIN
                        0\ADD := ACC ← ACC + MEM[EA],
                        1\SUB := ACC ← ACC - MEM[EA],
                        2\STR := MEM[EA] ← ACC,
                        3\LOAD := ACC ← MEM[EA],
                        4\SKPN := IF ACC<15> => PC ← PC + 1,
                        5\JMP := PC ← EA,
                        6 := NO.OP (),
                        7 := NO.OP ()
                        END
                END

        !! Execution.Cycle !!
        CYCLE[MAIN] := BEGIN
                PC←1 NEXT                ! program counter init
                REPEAT
                        BEGIN
                        XFETCH() NEXT    ! call fetch algorithm
                        EXEC()           ! call execution algorithm
                        END
                END
        END
```

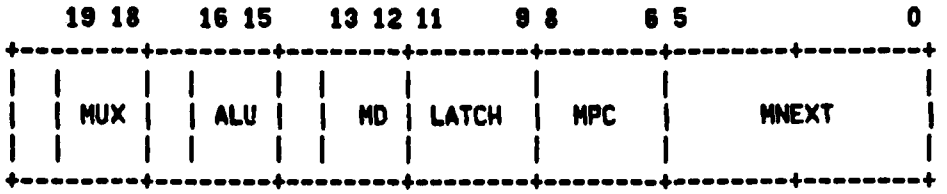
Figure 3-1: ISPS description of the TARGET machine

```
((SD (pre: (.MacroPC)=MAIN)
     (mod: TInvReg MacroPC PC)
     (env:)
     (post: #MacroPC=XFETCH #PC=1(12)))
 (SD (pre: (.MacroPC)=XFETCH)
     (mod: TInvReg MacroPC IR)
     (env:)
     (post: #MacroPC=FLOOP #IR=(DOT (WORDS MEM .PC .PC)
 (SD (pre: (.MacroPC)=FLOOP
           (NZEROP (USEQL (DOT (BITS IR 12))
                          0)))
     (mod: TInvReg MacroPC PC)
     (env:)
     (post: #MacroPC=EXEC #PC=(BITPLUS .PC 1(12))))
 (SD (pre: (.MacroPC)=EXEC
           (NZEROP (USEQL (DOT (BITS IR (PAIR 15 13)))
                          0)))
     (mod: TInvReg MacroPC ACC)
     (env:)
     (post: #MacroPC=XFETCH #ACC=(BITPLUS
              .ACC
              (DOT (WORDS MEM (USSUB .IR 11 0)
                              (USSUB .IR 11 0)
 (SD (pre: (.MacroPC)=EXEC
           (NZEROP (USEQL (DOT (BITS IR (PAIR 15 13)))
                          1)))
     (mod: TInvReg MacroPC ACC)
     (env:)
     (post: #MacroPC=XFETCH #ACC=(BITPLUS
              .ACC
              (BITMINUS (DOT (WORDS MEM
                                    (USSUB .IR 11 0)
                                    (USSUB .IR 11 0)
(SD (pre: (.MacroPC)=EXEC
          (NZEROP (USEQL (DOT (BITS IR (PAIR 15 13)))
                         2)))
    (mod: TInvReg MacroPC
          (WORDS MEM (DOT (BITS IR (PAIR 11 0)
    (env:)
    (post: #MacroPC=XFETCH #(WORDS MEM
                                   (USSUB .IR 11 0)
                                (USSUB .IR 11 0))=(.ACC)))
 (SD (pre: (.MacroPC)=EXEC
           (NZEROP (USEQL (DOT (BITS IR (PAIR 15 13)))
                          3)))
     (mod: TInvReg MacroPC ACC)
     (env:)
     (post: #MacroPC=XFETCH #ACC=(DOT (WORDS MEM
                                             (USSUB .IR 11 0)
                                             (USSUB .IR 11 0)
```

Figure 3-2: The SD description of the TARGET

```
(SD [pre: (.MacroPC)=EXEC
          (NZEROP (USEQL (DOT (BITS IR (PAIR 15 13)))
                         4))
          (NZEROP (DOT (BITS ACC 15]
    (mod: TInvReg MacroPC PC)
    (env:)
    (post: #MacroPC=XFETCH #PC=(BITPLUS .PC 1(12))))
(SD [pre: (.MacroPC)=EXEC
          (NZEROP (USEQL (DOT (BITS IR (PAIR 15 13)))
                         4))
          ~(NZEROP (DOT (BITS ACC 15]
    (mod: TInvReg MacroPC)
    (env:)
    (post: #MacroPC=XFETCH))
(SD (pre: (.MacroPC)=EXEC
          (NZEROP (USEQL (DOT (BITS IR (PAIR 15 13)))
                         5)))
    (mod: TInvReg MacroPC PC)
    (env:)
    (post: #MacroPC=XFETCH #PC=(USSUB .IR 11 0)))
(SD (pre: (.MacroPC)=EXEC
          (NZEROP (USEQL (DOT (BITS IR (PAIR 15 13)))
                         6)))
    (mod: TInvReg MacroPC)
    (env:)
    (post: #MacroPC=XFETCH))
(SD (pre: (.MacroPC)=EXEC
          (NZEROP (USEQL (DOT (BITS IR (PAIR 15 13)))
                         7)))
    (mod: TInvReg MacroPC)
    (env:)
    (post: #MacroPC=XFETCH))
(SD (pre: (.MacroPC)=FLOOP
          (NZEROP (USEQL (DOT (BITS IR 12))
                         1)))
    (mod: TInvReg MacroPC IR)
    (env:)
    (post: #MacroPC=FLOOP #IR=(USCONC
             (USSUB .IR 15 13)
             (USSUB (DOT (WORDS MEM (USSUB .IR 11 0)
                                    (USSUB .IR 11 0)))
                    12 0]
```

Figure 2. (continued)



The HOST machine (see schematic in Figure 3-3) includes the following: two memories, STORE, and ROM; registers R1, R2, R3, MAD, MPC (microprogram counter) and MI (microinstruction register); combinational circuits ALU, MD, and MUX; data paths; the scanner. R1 holds the value from the ALU that receives its value either from STORE or from R1; R2 holds the value from R3 or increments its old value; R3 holds the value from MD that receives its value from STORE or R3; MAD holds the value from MUX that receives its value either from R2 or R3.

The HOST repeats the cycle of loading the microinstruction register from the location in ROM that the microprogram counter points to; incrementing the microprogram counter; and scanning the microinstruction and decoding a field at a time. The scanner sends signals that establish data paths and latch values into registers. It also receives values from registers.

The precise ISPS description of the HOST machine is shown in Figure 3-4, and the description of the ROM in Figure 3-5. The description of the HOST includes the following declarations: the memories; the registers; the combinational logic; and the execution cycle that fetches and scans the IR. The microprogram is specified as a set of assignments to ROM. The comment in each assignment shows the microinstruction in a mnemonic form: The nonzero fields of each microinstruction are separated by @. The mnemonics correspond to the ones in the DECODE statements in Figure 3-4. For example,

MUXR3@LMAD@ONIND@10 means that MUX = 3, ALU = 0, MD = 0, LATCH = 6, MPC = 2 and MNEXT = 10.

The first phase of the proof converts the ISPS description of the HOST into a single SD whose *post:* field includes the complete representation of the HOST. This SD is used in

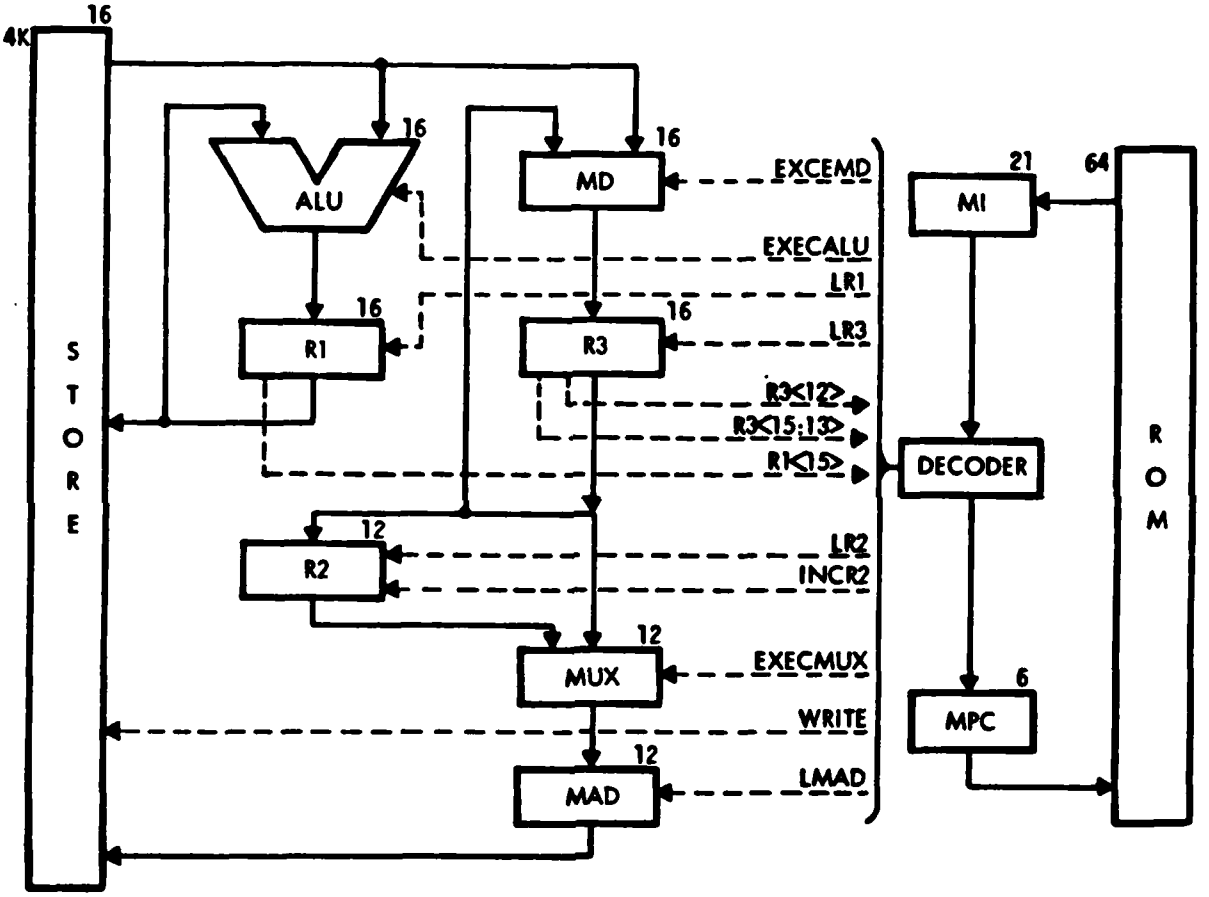


Figure 3-3: Schematic of the TOY Host

```
HOST := BEGIN

        ** Memory **
        ROM[0:63]<20:0>,
        STORE[0:4k]<15:0>

        ** Registers **
        MPC<5:0>,                   ! micro program counter
        MI<20:0>,                   ! micro instruction register
        MNEXT<5:0> := MI<5:0>,      ! next micro instruction
        R1<15:0>,                   ! Accumulator
        R2<11:0>,                   ! Program Counter
        R3<15:0>,                   ! Instruction Register
        MAD<11:0>                   ! memory address

        ** Combinational.Circuits **
        ALU<15:0>,                  ! arithmetic, logic unit
        MUX<11:0>,                  ! memory address multiplexer
        MD<15:0>                    ! memory data multiplexer

        ** Execution.Cycle **
        CYCLE[MAIN] := BEGIN
                REPEAT
                BEGIN
                MI ← ROM[MPC] NEXT
                MPC ← MPC + 1
                NEXT
                DECODE MI<19:18> =>
                        BEGIN
                        0 := NO.OP (),
                        1 := NO.OP (),
                        2\MUXR2 := MUX ← R2<11:0>,
                        3\MUXR3 := MUX ← R3<11:0>
                        END NEXT

                DECODE MI<16:15> =>
                        BEGIN
                        0 := NO.OP (),
                        1\ALUNOP := ALU ← STORE[MAD],
                        2\ALUADD := ALU ← R1 + STORE[MAD],
                        3\ALUSUB := ALU ← R1 - STORE[MAD]
                        END NEXT

                DECODE MI<13:12> =>
                        BEGIN
                        0 := NO.OP (),
                        1 := NO.OP (),
                        2\ALL := MD ← STORE[MAD],
                        3\ADD := MD ← R3<15:13>  STORE[MAD]<12:0>
                        END NEXT
```

Figure 3-4: ISPS description of the HOST

```
        DECODE MI<11:9> =>
                BEGIN
                0 := NO.OP (),
                1\LR1 := R1 ← ALU,
                2\LR2 := R2 ← R3<11:0>,
                3\LR3 := R3 ← MD,
                4\INCR2 := R2 ← R2 + 1,
                5\WRITE := STORE[MAD] ← R1,
                6\LMAD := MAD ← MUX,
                7\INIT := R2 ← 1
                END NEXT

        DECODE MI<8:6> =>
                BEGIN
                0 := NO.OP (),
                1\ONPOS := IF NOT R1<15> => MPC ← MNEXT,
                2\ONIND := IF R3<12> => MPC ← MNEXT,
                3 := NO.OP (),
                4\NXT := MPC ← MNEXT,
                5 := NO.OP (),
                6 := NO.OP (),
                7\ONOP := MPC ← R3<15:13>
                END
        END
        END

END
```

Figure 4. (continued)

```
ROM :=
        BEGIN

        !** Memory **!

        ROM[0:63]<20:0>

        !** Execution.Cycle **!

        CYCLE[MAIN] :=
                BEGIN
                ROM[0] ← #0201410 ;      ! ALUADD⊙LR1⊙NXT⊙8
                ROM[1] ← #0301410 ;      ! ALUSUB⊙LR1⊙NXT⊙8
                ROM[2] ← #0005410 ;      ! WRITE⊙NXT⊙8
                ROM[3] ← #0101410 ;      ! ALUNOP⊙LR1⊙NXT⊙8
                ROM[4] ← #0000416 ;      ! NXT⊙14
                ROM[5] ← #0002410 ;      ! LR2⊙NXT⊙8
                ROM[6] ← #0000410 ;      ! NXT⊙8
                ROM[7] ← #0000410 ;      ! NXT⊙8
                ROM[8] ← #2006000 ;      ! FETCH: MUXR2⊙LMAD
                ROM[9] ← #0023413 ;      ! ALL⊙LR3⊙NXT⊙11
                ROM[10] ← #0033000 ;     ! ADD⊙LR3
                ROM[11] ← #3006212 ;     ! FLOOP: MUXR3⊙LMAD⊙ONIND⊙10
                ROM[12] ← #0004000 ;     ! EXEC: INCR2
                ROM[13] ← #0000700 ;     ! ONOP⊙0
                ROM[14] ← #0000110 ;     ! ONPOS⊙8
                ROM[15] ← #0004410 ;     ! INCR2⊙NXT⊙8
                ROM[16] ← #0007410       ! INIT⊙NXT⊙8
        NEXT EXEC := NO.OP ()
                END
        END
```

Figure 3-5: The specification of the Microcode

the next section as the specification of the control state of the HOST in the mapping. The ISPS description of the microcode is converted to SD notation too.

The current implementation requires that the ISPS description of the HOST consist of a single cycle, for reasons of simplicity. The HOST will indeed usually be a single cycle because it represents hardware. Minor implementation changes will accommodate arbitrary ISPS descriptions of the HOST.

The next section introduces the mapping and the following section explains how the symbolic simulation of the TARGET by the microprogrammed HOST machine is set up and performed.

Relating the TARGET and the HOST

In order to show that one machine simulates another, a relation between the two must be established. The relation addresses control issues and data issues. The control part of the relation specifies all the pairs of control states (in the TARGET and HOST, respectively) that have the following properties: whenever a control state is reached in one machine then the corresponding one is reached in the other machine. Two obvious pairs are the pair of initial states and the pair of final states. A necessary condition for simulation (of terminating machines) is that corresponding initial states always lead to corresponding final states. The data part of the relation specifies the pairs of carriers that should have the same contents whenever a pair of control states is reached. This data relation is called a covering.

The control states in the TARGET machine to be mapped from or to were selected as the set of all the markings. For the particular TOY machine example the following markings were selected: the initial state is MAIN; the top of the main cycle is XFETCH; the infinite fetch loop is broken at FLOOP; the fetch algorithm is separated from the execution algorithm at all the control states in the TARGET map to or from a state described by the top of cycle of the HOST and an additional predicate (usually the value of the microprogram counter).

The top of Figure 3-6 shows a set of control relations; the first element of each is a marking (represented by an ISPS label) in the TARGET and the rest is a predicate that

together with the code of the HOST makes up its control state. The bottom of Figure 3-6 shows the coverings that specify the relation between registers (or memories) in the TARGET to registers (or memories) in the HOST.

During the first phase of the proof, a set of internal MAPPING records is generated from the concise representation of Figure 3-6. Figure 3-7 shows two out of the eight mappings. A MAPPING record has three fields: *from:*, that specifies the control state of either the TARGET or the HOST; *to:*, that specifies the corresponding control state of the other machine; and *map:*, that specifies the covering. The notion of MAPPING records is built into the SD proofchecker and is used in the second phase.

We have described the TARGET, the HOST+microcode, and the relation between them in three forms: English, formal, and a form that can be processed by the SD proofchecker. The first phase of the proof generated the batch of SD commands from the formal descriptions.

Symbolic Simulation

The previous sections presented the TARGET machine, the HOST machine with its microprogram, and the mapping between the machines. This section shows how the proof of simulation of the TARGET by the HOST with respect to the mapping was performed using the SD command batch. The simulation is performed within the state delta symbolic execution framework, thus it is called symbolic simulation.

The SD proof system operates by maintaining a "current state" of the execution, which can be manipulated by opening or closing proofs, or by applying SDs or mappings. A SD is a notation for specifying a segment of execution, either as the "goal" or for changing the current state. A SD has 4 fields: *pre:*, *mod:*, *env:*, and *post:*. When a SD is used to *Open* a proof, then the *pre:* is added to the current state and the *post:* becomes the goal; when it is being "applied", then the *pre:* must be true in the current state, and the effect of the SD is removing from the current state everything that depends on *mod:* and adding *post:*. A MAPPING has three fields: *from:*, *to:*, and *map*. When a mapping is "applied", its *from:* must be true in the current state, and the effect of the mapping is adding *to:* and *map:* to the current state.

```
((MAIN (.MPC)=16)
 (XFETCH (.MPC)=8)
 (FLOOP (.MPC)=11)
 (EXEC (.MPC)=13 (.MAD)=(USSUB .R3 11 0)))



((Covering MEM <<STORE 16 16>>)
 (Covering PC <<R2 12>>)
 (Covering ACC <<R1 16>>)
 (Covering IR <<R3 16>>)
 (Covering MacroPC <<MicroPC 2> <MPC 6>>)
 (Covering HInvReg <<MI 21> <MAD 12> <ALU 16> <MUX 12> <MD 16>>)
 (Covering TInvReg <<HInvReg 22>>))
```

Figure 3-6: Mapping between TARGET and HOST

```
(MAPPING [from: (.MPC)=11
                (SD (pre:)
                    (mod: MicroPC MI)
                    (env: MicroPC)
                    (post: #MI=(DOT (WORDS ROM .MPC))
                           (TR ((SEQ (USSET MPC $)
                                     (DECODE $ $ $ $ $)
                                     (DECODE $ $ $ $ $)
                                     (DECODE $ $ $ $ $)
                                     (DECODE $ $ $ $ $ $ $ $
                                             $)
                                     (DECODE $ $ $ $ $ $ $ $
                                             $))
                                (REPEAT $)
                                (ProcMark HOST]
         [to: (.MacroPC)=FLOOP)
         (map: (.MEM)=(.STORE)
               (.PC)=(.R2)
               (.ACC)=(.R1)
               (.IR)=(.R3)))
(MAPPING [from: (.MacroPC)=EXEC)
         [to: (.MPC)=13 (.MAD)=(USSUB .R3 11 0)
              (SD (pre:)
                  (mod: MicroPC MI)
                  (env: MicroPC)
                  (post: #MI=(DOT (WORDS ROM .MPC))
                         (TR ((SEQ (USSET MPC $)
                                   (DECODE $ $ $ $ $)
                                   (DECODE $ $ $ $ $)
                                   (DECODE $ $ $ $ $)
                                   (DECODE $ $ $ $ $ $ $ $ $)
                                   (DECODE $ $ $ $ $ $ $ $ $))
                              (REPEAT $)
                              (ProcMark HOST]
         (map: (.STORE)=(.MEM)
               (.R2)=(.PC)
               (.R1)=(.ACC)
               (.R3)=(.IR)))
```

Figure 3-7: Two of the MAPPING records

Figure 3-8 shows an outline of the batch of commands that drives the proof in the second phase. The first *Open* and *NewDecomposition* declare the memories and registers in the HOST machine. The *pre:* of the second *Open* includes the microcode and the mapping between the TARGET and the HOST. The *post:* of the same command includes the set of SDs that describes the TARGET machine. Executing this command adds the microcode and mapping to the current state and makes the TARGET the "goal". A sequence of seven *NewComposition* commands declares the memories and registers in the TARGET machine and their relation to the places in the HOST. The command *SymSimulate* performs the symbolic simulation according to a heuristic that we have developed.

The SymSimulate command executes a heuristic that drives the symbolic simulation. For each SD in the "goal" do the following: open the SD; apply a mapping from the TARGET to the HOST; symbolically execute (i.e., keep applying SDs) until the state can be mapped back to the TARGET; apply the mapping to the TARGET; close the SD. Finally close the whole "goal".

The combined effect of the two phases of the proof is the generation of a set of SDs from the TARGET using symbolic execution of the TARGET and proving these SDs by using symbolic execution of the HOST and microcode. The rest of the effort is setting up the right relations among the registers and memories and between the HOST and TARGET to assure integrity of the proof. Note that the only input needed is the ISPS description of the TARGET, HOST, and ROM and the concise representation of the mapping between the machines. The rest is done automatically.

3.2 THE FTSC

The FTSC was chosen as the real example on which to try out the microcode verification system because it is a general-purpose computer with enough features to thoroughly test the system; in addition, it is still in the development stage, so that successful verification or discovery of bugs would influence the final version.

Some of the characteristics of the FTSC (as of May 1979) are:

```
((Open (vars: MicroPC EXP MD MUX ALU MAD R3 R2 R1 MI MPC STORE ROM UNDEFINED
            CLKLOC& LABLOC& ASSLOC& ARRLOC&)
       (SD (pre: (Covering OMEGA
                           <<MicroPC 1> <EXP 44Q> <MD 16> <MUX 12>
                             <ALU 16> <MAD 12> <R3 16> <R2 12> <R1 16>
                             <MI 21> <MPC 6> <STORE 16 10001Q>
                             <ROM 21 100Q> <UNDEFINED 44Q> <CLKLOC& 44Q>
                             <LABLOC& 44Q> <ASSLOC& 44Q> <ARRLOC& 44Q>>))
           (mod: OMEGA)
           (env:)
           (post:)))
 (NonDecomposition (Covering OMEGA
                           <<MicroPC 1> <EXP 44Q> <MD 16> <MUX 12>
                             <ALU 16> <MAD 12> <R3 16> <R2 12> <R1 16>
                             <MI 21> <MPC 6> <STORE 16 10001Q>
                             <ROM 21 100Q> <UNDEFINED 44Q> <CLKLOC& 44Q>
                             <LABLOC& 44Q> <ASSLOC& 44Q> <ARRLOC& 44Q>>))
 (Open (vars: MicroPC EXP IR ACC PC MEM UNDEFINED CLKLOC& LABLOC& ASSLOC&
            ARRLOC&)
       (SD (pre: (DOT (WORDS ROM 0))=(OCONST 2014100 21)

        .... ||| Specification of microcode |||

                (MAPPING (from: (.MacroPC)=MAIN)
                         (to: (.MPC)=16
                              (SD (pre:)
                                  (mod: MicroPC MI)
                                  (env: MicroPC)
                                  (post: #MI=(DOT (WORDS ROM .MPC))
                                         (TR ((SEQ (USSET MPC $)
                                                   (DECODE $ $ $ $ $)
                                                   (DECODE $ $ $ $ $)
                                                   (DECODE $ $ $ $ $)
                                                   (DECODE $ $ $ $ $ $ $ $ $)
                                                  (DECODE $ $ $ $ $ $ $ $ $))
                                             (REPEAT $)
                                             (ProcMark HOST)
                         (map: (.STORE)=(.MEM)
                               (.R2)=(.PC)
                               (.R1)=(.ACC)
                               (.R3)=(.IR)))

        .... ||| All mappings |||
```

Figure 3-8: Outline of the command batch

```
            (mod:)
            (env:)
            (post: (SD (pre: (.MacroPC)=MAIN)
                       (mod: TInvReg MacroPC PC)
                       (env:)
                       (post: #MacroPC=XFETCH #PC=1(12)))

        .... ((( State Delta representation of TARGET )))

(RenComposition (Covering MEM <<STORE 16 16>>))
(RenComposition (Covering PC <<R2 12>>))
(RenComposition (Covering ACC <<R1 16>>))
(RenComposition (Covering IR <<R3 16>>))
(RenComposition (Covering MacroPC <<MicroPC 2> <MPC 6>>))
(RenComposition (Covering HInvReg
                          <<MI 21> <MAD 12> <ALU 16> <MUX 12> <MD 16>>))
(RenComposition (Covering TInvReg <<HInvReg 22>>))
(SymSimulate))
```

Figure 8. (continued)

- 112 instructions, including integer, floating point, and vector operations
- data formats: fixed point (32-bit, two's complement integer) and floating point (24-bit, two's complement mantissa; 8-bit, two's complement exponent)
- 9 address modes
- 8 general-purpose registers (that serve as accumulators, index registers, or address pointers) and 8 working registers
- 10 interrupt levels
- 61K of addressable program memory

The first step in the verification process is writing the formal host and target machine descriptions in ISPS. Ideally, the designer of the machine would write the formal description along with the informal description ("user's manual"). In lieu of this, the writer of the formal descriptions must submit them to the designer for "description verification" (that this is really the machine informally described in the manual) before proceeding with the proof. In addition, the writer of the formal descriptions may discover "bugs" (inconsistencies or incompleteness) in the user manual. As a formal description is being written, its writer will probably be in need of information which was either omitted from the machine user manual or presented there in an ambiguous or contradictory way.

Our experience yielded approximately 120 questions on the documentation, accumulated over a period of about six months. Approximately 80 answers were finally obtained from various persons who had "inside" information about the construction of the FTSC. Typical difficulties are missing information, multiple names for the same value, e.g., AMODE and AM, and inconsistencies between written and diagrammed specifications.

As explained earlier, we consider the total problem of microcode verification as consisting of two parts: the proof that the host machine with its microcode implements the target machine (as described in a language containing only those operations available to the host) and the proof that the target machine, instruction by instruction, satisfies some higher level specification. For example, the target machine description of

the integer multiply and divide instructions, and all floating point instructions, would most likely consist of an algorithm using the host machines operations of shifting, testing, adding, XORing, etc. The higher level specification would be that these instructions do in fact find the product, quotient, etc. to a given precision. The instruction definitions given in the user manual, which are largely English, are most likely those instructions needing this second level of proof.

All of our work to date on the verification of the FTSC has been concerned with the step from the target to the higher specification. This seemed a wise choice, since we knew that at the start of our project the FTSC host machine design was not finalized, although the target machine would remain more or less the same. In addition, many aspects of the system had to be developed before a truly large example could be attacked.

The particular instruction chosen was square root. Square root was chosen because of the relative compactness of its algorithmic description in the target machine, and the wide difference between the algorithm and its higher specification. Although the second-level verification has nothing to do with the microcode or the host machine, one characteristic making it less than general program verification is that the data types used in the target and higher level descriptions are usually restricted to be bitstrings and integers in the target, and values of bitstrings and reals in the higher level. Thus we used the square root instruction as a testing ground for developing the automatic simplification of expressions in these data types.

The status of our work on the square root algorithm is that the simplifier is able to handle automatically all the derivations needed to complete the proof of correctness. Smoothing the user interface and gracefully setting up the induction needed for the loop remain to be done.

It is hoped that many of the special simplification rules adopted in proving the square root will also be useful in the other proofs of higher level correctness.

Square Root Proof

In this section we give the ISPS version of the algorithm that constitutes the FTSC target machine description of the floating point square root instruction (SRTF). See

Figure 3-9. This description of the algorithm was written on the basis of the microcode flowchart, which is derived directly from the host description and the microcode. Then we show the derivations the simplifier is able to accomplish automatically in proving that SRTF finds the square root to within a certain accuracy.

Let us "talk through" the algorithm now: The first line decides if the input is to be from register GPXRA or register MD. If the input is negative, the algorithm is terminated with overflow flag set. If the input is 0, the algorithm is terminated with output register GPXRB set to the floating representation of 0. From here on the algorithm splits into two parts: the calculation of the new exponent and the calculation of the new mantissa. The exponent calculation splits depending on whether it is even or odd. If the old value is even, the new exponent is half the old value. If the old value is odd, it is made even by adding 1 and shifting the mantissa accordingly (in the even case the mantissa is shifted two bits; in the odd case, only one bit). Now the new value is half the old value (with a check for exponent overflow thrown in). The mantissa is now calculated by a variation of the longhand high school square root algorithm. The mantissa is shifted two bits at a time through the loop 23 times. The loop has two branches according to the sign of the "remainder," the register SUM.

The theorem which expresses the correctness of SRTF is

Theorem: If $FL(INPUT)=x\geq 0$, then SRTF terminates with $FL(OUTPUT)^2 \leq x \leq FL^+(OUTPUT)^2$.

If $FL(INPUT)<0$, then SRTF terminates with OVFF=1.

Explanation of notation: FL(R) is the value of the bitstring R as a floating point number in the FTSC format: 24 leftmost bits coding two's complement fractional mantissa and rightmost 8 bits coding two's complement exponent. INPUT is either the register GPXRA or MD, depending on AMODE. OUTPUT is the register GPXRB. $FL^+(R)$ is floating successor to FL(R), i.e.,

$$FL^+(R) = (TCVAL(R\langle 31:8\rangle)+1) * 2^{TCVAL(R\langle 7:0\rangle)-23}.$$

Letting $MAN(R) = TCVAL(R\langle 31:8\rangle) * 2^{-23}$ and $EXP(R) = TCVAL(R\langle 7:0\rangle)$, it is sufficient to prove

```
SRTF:=

        BEGIN
        DECODE AMODE=>(W0←W1←GPXRA,W0←W1←MD) NEXT
        IF W0 LSS 0=>(OVFF←1 NEXT LEAVE SRTF) NEXT
        IF W0<31:8> EQL 0=>(GPXRB←"80 NEXT LEAVE SRTF) NEXT
        W0<31:8>←W0<31:8> SL0 1 NEXT
        W0<7:0>←0 NEXT
        DECODE W1<0>=>
                BEGIN
                0:=(GPXRB←W0<31:30> NEXT
                   W0←W0 SL0 2  NEXT
                   W1<31:8>←0   NEXT
                   W1<7:0>←W1<7>@W1<7:1> ),
                1:=(GPXRB←W0<31> NEXT
                   W0←W0 SL0 1 NEXT
                   W1<31:8>←0 NEXT
                   EXPOUT←W1<7>@W1<7:0> + 1 NEXT
                   W1<7:0>←EXPOUT<7:0> NEXT
                    W1<7:0>←W1<7>@W1<7:1> NEXT
                   IF EXPOUT<8> XOR EXPOUT<7>=>W1<7:0>←#100 )
                END
        NEXT
        SUM←GPXRB-1 NEXT
        GPXRB←SUM<29:0>@W0<31:30> NEXT
        COUNTER←0  NEXT
        SLOOP:=
                REPEAT
                BEGIN
                COUNTER←COUNTER+1 NEXT
                W0<31:8>←W0<31:8> SL0 2 NEXT
                DECODE SUM<31>=>
                BEGIN
                0:= (W1<31:8>←2*W1<31:8> + 1 NEXT
                        IF COUNTER EQL 23=>(LEAVE SLOOP) NEXT
                        W2←4*W1<31:8>  + 1 NEXT
                        SUM←GPXRB-W2 NEXT
                        GPXRB←SUM<29:0>@W0<31:30>),
                1:= (W1<31:8>←2*W1<31:8>  NEXT
                        IF COUNTER EQL 23=>(LEAVE SLOOP) NEXT
                        W2←4*W1<31:8> + 3 NEXT
                        SUM←GPXRB+W2 NEXT
                        GPXRB←SUM<29:0>@W0<31:30>)
                END
                END
        NEXT
        GPXRB←W1
        END
```

Figure 3-9: ISPS description of the square root algorithm

(I) If EXP(INPUT)=e is even and MAN(INPUT)*2^{46}=ARG, then SRTF terminates with 2*EXP(OUTPUT)=e and (MAN(OUTPUT)*2^{23})^2 ≤ ARG ≤ (MAN(OUTPUT)*2^{23}+1)^2, and

(II) If EXP(INPUT)=e is odd and MAN(INPUT)*2^{45} = ARG, then SRTF terminates with 2*EXP(OUTPUT)=e+1 and (MAN(OUTPUT)*2^{23})^2 ≤ ARG ≤ (MAN(OUTPUT)*2^{23}+1)^2.

So the proof is carried out by

(1) symbolically executing through the end of the exponent calculation for even and odd input exponent, and proving the relevant parts of (I) and (II) at that point (note that OUTPUT is assigned the contents of working register W1 at the end of SRTF);

(2) at that point, for even input exponent,

$$MAN(INPUT)*2^{46} = USVAL(GPXRB\langle 1:0\rangle @W0\langle 31:10\rangle)*2^{22} = ARG,$$

and for odd exponent,

$$MAN(INPUT)*2^{45} = ARG.$$

Thus to complete both (I) and (II) it remains to show that

CLAIM: TCVAL(OUTPUT<31:8>)^2 ≤ ARG ≤TCVAL(OUTPUT<31:8>+1)^2.

Here is where we use induction to prove loop invariants that lead to a proof of the CLAIM. Let R_i denote the contents of R after i times through the loop, that is, the last contents before COUNTER changes from i to i+1.

The CLAIM is proved from

SUBCLAIM: For 1≤i≤23, USVAL($W1_i$<30:8>)^2 ≤ int(ARG*2^{2i-46}) ≤ (USVAL($W1_i$<30:8>)+1)^2.

(The actual calculation with the integer part function int is done by noting that if X=USVAL(R), then int(X*2^{-k}) = USVAL(R SR0 k).)

The CLAIM is proved from the SUBCLAIM by taking i=23. The SUBCLAIM is implied by the first three of the following loop invariants for 1≤i≤22. ((H1) is shown here for the case of even exponent only).

(H1) $(2*USVAL(W1_i\langle 30:8\rangle)+1)^2 + TCVAL(SUM_i) = USVAL(a\langle 30:8\rangle @0(23)\ SR0\ 44-2i)$

(H2) $TCVAL(SUM_i) \leq 4*USVAL(W1_i\langle 30:8\rangle) + 2$

(H3) $-TCVAL(SUM_i) \leq 4*USVAL(W1_i\langle 30:8\rangle) + 1$

(H4) $W0_i =_{US} (a\langle 28:8\rangle @0(11)\ SL0\ 2i)$

(H5) $W1_i\langle 31:i+8\rangle =_{US} 0(24-i)$

(H6) $W2_i\langle 31:i+2\rangle =_{US} 0(30-i)$

(H7) $SUM_i\langle 29:0\rangle =_{US} GPXRB_i\langle 31:2\rangle$

(H8) $SUM_i =_{TC} GPXRB_i\langle 31:2\rangle$

(H9) $GPXRB_i\langle 1:0\rangle =_{US} W0_i\langle 31:30\rangle$

Thus we prove that if (H1)-(H9) are true for $1 \leq i \leq 21$, then they are true for i+1. Additional induction hypotheses ((H4)-(H9) were found to facilitate the proof of (H1)-(H3)). Then we prove that if the SUBCLAIM is true for $1 \leq i \leq 22$, then it is true for i+1. The simplifier automatically carries out these deductions.

The following is the batch containing the proof of the square root algorithm as it is read into MICHOVER in form to be automatically checked.[1]

```
(BATCHSQRT
 [(InitProof SQRTM)
  (InstantiateContents GPXRA a)
  (Prove
   [SD (pre: (.AMODE)=0 (TCGEQ (USSUB a 31 8)
                              0)
          (TCNEQ (USSUB a 31 8)
               0)
          (USEQL (USSUB a 0 0)
               0)
```

[1] Actually, in the present form of the system the INVARIANT and LABEL must be given in expanded form at every occurrence.

```
        (SD (pre: (NZEROP (USEQL .AMODE 0)))
            (mod: MicroPC)
            (env: MicroPC)
            (post: @Program)))
      (mod: OMEGA)
      (env: GPXRA)
      (post: (NZEROP (REALEQUAL (PRODUCT (EXPVAL #GPXRB)
                                         2)
                                (EXPVAL a)))
           [NZEROP (REALLEQ (POWER (PRODUCT (MANVAL #GPXRB)
                                            (POWER 2 23))
                                   2)
                            (PRODUCT (MANVAL a)
                                     (POWER 2 56Q]
           (NZEROP (REALLEQ (PRODUCT (MANVAL a)
                                     (POWER 2 56Q))
                            (POWER (REALPLUS (PRODUCT (MANVAL #GPXRB)
                                                      (POWER 2 23))
                                             1)
                                   2]
((ProposeMode (.COUNTER)=1)
 [ProvebyCases [SD (pre:)
                   (mod: OMEGA)
                   (env: OMEGA)
                   (post: #COUNTER=(1@Invariant)
                        (SD (pre:)
                            (mod: MicroPC COUNTER)
                            (env: MicroPC)
                            (post: #COUNTER=(USSUB (TCPLUS .COUNTER 1)
                                                   31 0)@Label]
                 (((USSUB .SUM 31 31)=1
                  ([ProposeMode ((.COUNTER)=1
                                 and (SD (pre:)
                                          (mod: MicroPC COUNTER)
                                          (env: MicroPC)
                                          (post: #COUNTER=(USSUB
                                                 (TCPLUS .COUNTER 1)
                                                 31 0)@Label]
                   (Close)))
                  ((USSUB .SUM 31 31)=0
                   ([ProposeMode ((.COUNTER)=1
                                 and (SD (pre:)
                                          (mod: MicroPC COUNTER)
                                          (env: MicroPC)
```

```
                                (post: #COUNTER=(USSUB
                                       (TCPLUS .COUNTER 1)
                                       31 0)@Label]
            (Close]
[ApplySD (SD (pre: ((USSUB .SUM 31 31)=1 or (USSUB .SUM 31 31)=0))
         (mod: OMEGA)
         (env: OMEGA)
         (post: (T or T)
              #COUNTER=1 @Invariant
              (SD (pre:)
                 (mod: MicroPC COUNTER)
                 (env: MicroPC)
                 (post: #COUNTER=(USSUB (TCPLUS .COUNTER 1)
                                  31 0)@Label]
(Prove [SD (pre:)
        (mod:)
        (env: OMEGA)
        (post: (.COUNTER)=1 @Invariant
             (SD (pre:)
                (mod: MicroPC COUNTER)
                (env: MicroPC)
                (post: #COUNTER=(USSUB (TCPLUS .COUNTER 1)
                                 31 0)@Label]
    ((ProposeMode)))
[ProvebyCases [SD (pre: (NZEROP (REALLEQ 1 .COUNTER))
                  (NZEROP (REALLEQ .COUNTER 21))@Invariant
                  (SD (pre:)
                     (mod: MicroPC COUNTER)
                     (env: MicroPC)
                     (post: #COUNTER=(USSUB (TCPLUS .COUNTER 1)
                                      31 0)@Label)))
              (mod: OMEGA)
              (env:)
              (post: #COUNTER=(REALPLUS .COUNTER 1)@Invariant
                   (SD (pre:)
                      (mod: MicroPC COUNTER)
                      (env: MicroPC)
                      (post: #COUNTER=(USSUB (TCPLUS .COUNTER 1)
                                       31 0)@Label]
          ((((USSUB .SUM 31 31)=1 and (USSUB .GPXRB 31 31)=1)
            ((ProposeMode)))
           (((USSUB .SUM 31 31)=0 and (USSUB .GPXRB 31 31)=0)
            ((ProposeMode]
(Prove [SD (pre: (NZEROP (REALLEQ 1 .COUNTER))
```

```
            (NZEROP (REALLEQ .COUNTER 21))@Invariant
            (SD (pre:)
               (mod: MicroPC COUNTER)
               (env: MicroPC)
               (post: #COUNTER=(USSUB (TCPLUS .COUNTER 1)
                                      31 0)@Label)))
         (mod: OMEGA)
         (env:)
         (post: #COUNTER=(REALPLUS .COUNTER 1)@Invariant
              (SD (pre:)
                 (mod: MicroPC COUNTER)
                 (env: MicroPC)
                 (post: #COUNTER=(USSUB (TCPLUS .COUNTER 1)
                                        31 0)@Label]
      ([ApplySD (SD (pre: ((USSUB .SUM 31 31)=1
                           and (USSUB .GPXRB 31 31)=1
                           or (USSUB .SUM 31 31)=0
                           and (USSUB .GPXRB 31 31)=0)
                      (NZEROP (REALLEQ 1 .COUNTER))
                      (NZEROP (REALLEQ .COUNTER 21))@Invariant
                      (SD (pre:)
                         (mod: MicroPC COUNTER)
                         (env: MicroPC)
                         (post: #COUNTER=(USSUB (TCPLUS .COUNTER 1)
                                                31 0)@Label)))
                 (mod: OMEGA)
                 (env:)
                 (post: (T or T)
                      #COUNTER=(REALPLUS .COUNTER 1)@Invariant
                      (SD (pre:)
                         (mod: MicroPC COUNTER)
                         (env: MicroPC)
                         (post: #COUNTER=(USSUB (TCPLUS .COUNTER
                                                        1)
                                                31 0)@Label]
         (Close)))
(PerformInduction (SD $)
                  (SD (&)
                     $))
(ProposeMode (SD (pre:)
                 (mod: MicroPC COUNTER)
                 (env: MicroPC)
                 (post: #COUNTER=(USSUB (TCPLUS .COUNTER 1)
                                        31 0)@Label)))
```

```
(InstantiateContents W1 w1)
[ProvebyCases
 [SD (pre:)
    (mod: OMEGA)
    (env: OMEGA)
    (post: (NZEROP (REALEQUAL (PRODUCT (EXPVAL #GPXRB)
                                       2)
                              (EXPVAL a)))
           [NZEROP (REALLEQ (POWER (PRODUCT (MANVAL #GPXRB)
                                            (POWER 2 23))
                                   2)
                            (PRODUCT (MANVAL a)
                                     (POWER 2 56Q]
           (NZEROP (REALLEQ (PRODUCT (MANVAL a)
                                     (POWER 2 56Q))
                            (POWER (REALPLUS (PRODUCT (MANVAL #GPXRB)
                                                      (POWER 2 23))
                                             1)
                                   2]
  (((USEQL (USSUB .SUM 31 31)
           0)
    ((ProposeMode)))
   ((USEQL (USSUB .SUM 31 31)
           1)
    ((ProposeMode]
  (ProposeMode])
(BATCHSQRT)
```

4. CONCLUSIONS

PLANNED EXTENSIONS

The basic theoretical work for proofs of correctness of sequential microcode is reasonably complete, and a preliminary system for carrying out proofs has been built and exercised. Within the scope of the present work, the following extensions are planned.

Proof Language

The system is divided into a user interface and a rigorous proofchecker. In the present implementation, the user interface knows too little about the direction of the proof. In a proof by cases, for example, the separate cases are presented to the proofchecker, then combined. It is possible to declare the intended result in a superior proof, but no use is made of this information in either the user interface or the kernel.

We now see that the user interface can interpret a simple goal-oriented language. For a proof by cases, the user would specify what lemma is to be proven and would specify that the form of the proof is to be by cases with a given predicate. Room for specifying the details of each subproof would also exist, but the packaging of the separate proofs would be carried out by the proofchecker. In the present system, a proof by cases now looks like the following:

```
(Open P)
(Open P and C)
  <details of the proof of the first case>
(Close P and C)
(Open P and not C)
  <details of the proof of the second case>
(Close P and not C)
(CombineCases)
(Close P)
```

In many instances, the proof of each case may be carried out automatically. In the present system, a ProposeMode statement is required. We can eliminate the "obvious" proofs if we use null lists where proof details are permitted. Combined with the automatic setup and packaging of compound proofs, the proof above might become the following:

(Prove P (Cases C <room for details of positive subcase>
<room for details of negative subcase>))

Similar savings would result in proofs by induction. Some of the savings are not apparent from proof sketches like the ones above. The lemmas are often quite lengthy. Even with the lemma suppressed from the Close command, the current system requires three copies of the main lemma, one for the statement of the lemma in the main proof, and two more for the subcase proofs. The compressed form requires only one appearance of the lemma. In addition, the compressed form is much more readable and, we hope, more writable.

Editing

The present system permits only limited editing of the proof. Using the structured proofs illustrated above, it should be possible to edit a proof quite freely and have the proof restarted from the last point it was changed.

Efficiency

The present system is fairly slow. With a little experimentation, it has become clear that a lot of time is expended in the simplifier. The simplifier has evolved through an accretion process, and is due for a complete redesign. We have also studied Derek Oppen's work (see, for example, [Nelson and Oppen 78]), and it appears reasonable to use his simplifier for parts of the system. His simplifier is carefully crafted and should be much faster.

FUTURE CONSIDERATIONS

A number of ideas for logical next steps have emerged, though these are beyond the scope of the present effort.

Floating Point Arithmetic Specification

It is obvious that we must allow other floating point formats than that of the FTSC. The parameters needed to specify the format should be variables which can be set by the user to fit his particular application. In addition, floating point arithmetic needs to be characterized precisely. Notation to describe the intended precision of the results and relationship between floating point operations and the corresponding abstract operations

on the reals would materially reduce the size of the target machine description and remove the need for proving a separate set of constraints.

Some of the initial work has been done by Brown and others [Brown 77, Brown 78, Wijngaarden 64, Kahan 77a, Kahan 77b].

Timing

Performance characteristics play a large part in the design of host machines and in the design of the microcode. However, to date no work has been done to characterize the running time of microcode. Proofs of running time limits should be reasonably straightforward, but work is needed on the specifications.

Concurrency

Essentially no work has been done on correctness proofs of truly concurrent microcode. The present work requires a sequentialized model of the host and target machines. Extensions to the basic theory will be required to model concurrency.

REFERENCES

[Alfvin 79] Peter W. Alfvin, A Formal Definition of AMDL, Master's thesis, University of California, Los Angeles, 1979.

[Barbacci et al. 77] Mario R. Barbacci, Gary E. Barnes, Roderic G. Cattell, and Daniel P. Siewiorek, The ISPS Computer Description Language, 1977. (Unpublished paper from Carnegie-Mellon University.)

[Bell and Newell 71] Gordon C. Bell and Allen Newell, *Computer Structures: Readings and Examples*, McGraw-Hill, New York, 1971.

[Birman & Joyner 76] A. Birman and William H. Joyner, "A Problem-Reduction Approach to Proving Simulation Between Programs," *IEEE Transactions on Software Engineering* SE-2, (2), June 1976, 87-96.

[Brown 77] W. S. Brown, *A Realistic Model of Floating Point Arithmetic*, Bell Laboratories, Technical Report 58, 1977.

[Brown 78] W. Stanley Brown and Stuart I. Feldman, *Environment Parameters and Basic Functions for Floating-Point Computation*, Bell Laboratories, Technical Report 72, 1978.

[Burstall 74] R. M. Burstall, "Program Proving as Hand Simulation with a Little Induction," in *Information Processing 74*, pp. 308-312, North-Holland, Amsterdam, 1974.

[Crocker 77] Stephen D. Crocker, *State Deltas: A Formalism for Representing Segments of Computation*, Ph.D. thesis, University of California, Los Angeles, 1977.

[Joyner et al. 78] William H. Joyner Jr., William C. Carter, and Daniel Brand, "Using Machine Descriptions in Program Verification," in *Information Technology: Proceedings of the 3rd Jerusalem Conference on Information Technology (JCIT3)*, pp. 515-522, North-Holland, Amsterdam, 1978.

[Kahan 77a] W. Kahan and B. N. Parlett, *Can You Count on Your Calculator?*, University of California, Berkeley, Memorandum No. UCB/ERL M77/21, 1977.

[Kahan 77b] W. Kahan, *And Now for Something Completely Different: The Texas Instruments SR-52*, University of California, Berkeley, Memorandum No. UCB/ERL M77/23, 1977.

[London 77] Ralph L. London, "Perspectives on Program Verification," in Raymond T. Yeh (ed.), *Current Trends in Programming Methodology*, pp. 151-172, Prentice-Hall, 1977.

[Manna & Waldinger 78] Zohar Manna and Richard Waldinger, "Is 'Sometime' Sometimes Better than 'Always'?," *Communications of the ACM* 21, (2), February 1978, 159-172.

[Marcus 79] Leo Marcus, State Deltas that Remember: a System of Describing State Changes, 1979. (Submitted for publication.)

[Nelson and Oppen 78] C. G. Nelson and D. C. Oppen, *Simplification by Cooperating Decision Procedures*, Stanford University, CS Report No. STAN-CS-78-652, 1978. (AI Memo AIM311.)

[Patterson 77] David Patterson, *Verification of Microprograms*, Ph.D. thesis, University of California, Los Angeles, 1977.

[Raytheon Corp 79] Raytheon Corp., *Brassboard Fault Tolerant Spaceborne Computer (BFTSC)*, Raytheon Corp., Technical Report ER79-4135, May 1979.

[Wegbreit 77] Ben Wegbreit, "Constructive Methods in Program Verification," *IEEE Transactions on Software Engineering* SE-3, (3), May 1977, 193-209.

[Wijngaarden 64] A. van Wijngaarten, "Numerical Analysis as an Independent Science," *BIT* 6, 1964, 66-81.

Appendix A
THE SYSTEM

This appendix describes the operation of the proofchecker, the state delta expression language, and the simplifier.

A.1 PREPARING AND RUNNING A PROOF

The MICROVER system is a LISP program that is loaded from TOPS20 exec by typing <AMDSYS>MICROVER.EXE[2]. The program is started by the LISP function **StartExec**, and can be restarted by the function **ContinueExec**. Both functions put the system in **exec** mode, which provides a set of commands to prepare and run proofs.

The proof checker is driven by a sequence of proofsteps. Each proofstep is submitted one at a time to the kernel, which checks its applicability and updates the state of the proof according to the specific proofstep. Although the user is responsible for preparing the proofsteps, the MICROVER system provides various aids for preparing and submitting them. The most important aid is the the batch. The batch consists of a sequence of proofsteps that is submitted by MICROVER under user supervision.

A.1.1 Exec Mode

Exec mode provides several ways to prepare and submit proofsteps, as well as some miscellaneous tasks.

The following commands are used to prepare and submit proofs:

UserMode	This command puts the system in a mode that provides the user with convenient facilities to prepare individual proofsteps. In particular, it completes key-words, prompts with parameter names, etc. The proofsteps are prepared one at a time, and submitted immediately.
SaveTranscript	This command accumulates the successful proofsteps from the last session into a batch. The batch (in the form of a LISP function) can be stored away, submitted again, or otherwise manipulated.

[2] The system is currently available on the ISIE machine, accessible over the ARPANET.

BatchMode	This command controls the submitting of a batch. See below for more details.
FixLast	Lets the user edit and resubmit the proofstep that was last submitted. The full power of the INTERLISP editor is available. It is a convenient way to recover from an error.
GenBATCH	GenBATCH prepares a batch of proofsteps according to the ISPS descriptions of the target-machine, host-machine, ROM, and mapping. This command is used for symbolic simulation. Three TOPS20 files and two LISP variables must exist before executing GenBATCH: The description of the target, host and ROM should reside in the files TARG.ISP, HOST.ISP and ROM.ISP, respectively. The mapping should reside in the LISP variables MAPPING$LIST and COVERING$LIST. The result of GenBATCH is a list of proofsteps for submission in batch mode. The user is queried as to where to store the list.

The following miscellaneous commands are provided by exec mode:

ResetProof	Clears the whole proof, ready to begin a new session.
SetSwitch	Sets, resets, or checks the value of a trace switch.
DisplaySWLIST	Displays the value of all the trace switches.
DisplayState	Displays the current state of the proof.
DisplayLast	Displays the last proofstep that was submitted.
Quit	Returns the system to the LISP level.

A.1.2 BatchMode

Batchmode initializes and controls the submitting of a batch that exists as a TOPS20 file. This batch could be generated off line using an editor, by the SaveTranscript command, or by the GenBATCH command (see next section). It provides the following batch commands:

OpenBatch	Reads the batch from a file and initializes the batch-pointer to the first proofstep in the file.

DisplayNext	Displays the proofstep to which the batch pointer is pointing.
PerformNext	Submits the proofstep to which the batch-pointer is pointing and advances[3] it.
Doit	Performs a fixed number of proofsteps from the batch file. The user is asked for the number.
WholeBatch	Displays the complete list of proofsteps in the batch file last read by OpenBatch.
Quit	Returns to the exec mode.

A.2 BASIC PROOFSTEPS

The basic "proof action" that MICROVER uses is setting goal to sd:post, and advancing the current state until the goal becomes true. Using combinations of this proof action for the right state deltas can accomplish symbolic execution, symbolic simulation, proofs by cases, or proofs by induction.

MICROVER provides a data base to hold the current state and a kernel that processes a sequence of basic proofsteps. Before carrying out a proofstep, MICROVER checks that all of the requirements are satisfied. If they are not, an error message is printed and the proofstep is aborted with no change to the data base. The following basic proofsteps are available in the system:

A.2.1 Beginning and Ending a Proof

(Open vars-list sd) meaning: Initiates proof of sd.

arguments: sd is a state delta and vars-list is a list of places or variables.

requirements: The places in sd:mod and sd:env must be registered (see below).

effects: Creates a current state consisting of sd:pre and those

[3] In case of failure, the exec command DisplayLast and FixLast still points to the failed proofstep (and can be used for recovery)

predicates from the previous state whose support is contained in sd:env; creates a new goal of sd:post; the prior state of the database and the place graph are restored when the proof is complete, except that the proven state delta is added to the prior state. (See Close, below).

(Close)

meaning: Terminates the proof of the most recently Opened state delta (goal) assuming the postcondition of goal is true in the current state.

arguments: none

requirements: sd:post simplifies to true.

effects: Restores the proof system to its state prior to the most recent Open, with the addition of the proven state delta.

A.2.2 Registering Places

(NewDecomposition covering)

meaning: Registers new subplaces.

arguments: Covering is of the form (Covering place ((subplace length) ... (subplace length))).

requirements: Mother place must be registered; daughter places must not be registered.

effects: The place graph is extended with new covering relationship.

(NewComposition)

meaning: Registers new superplaces.

arguments: Covering as above.

requirements: Mother place must not be registered; daughter places must be registered and disjoint.

effects: The place graph is extended with new covering relationship.

A.2.3 Advancing the Computation

(ApplySD sd)

meaning: Advance the execution by applying sd.

arguments: sd is a state delta.

requirements: sd:pre must simplify to true in the current state, and sd:mod must be contained in the modification list for the most recently Opened state delta.

effects: Deletes from the current state all predicates supported by places in sd:mod, and adds sd:post.

A.2.4 Case Analysis and Loops

(CombineCases sd-list)

meaning: Combines the state deltas in sd-list into one state delta.

arguments: sd-list is a list of state deltas (sd_1 ... sd_n) where sd_i is of the form

(SD (pre: $case_i$
pred)
(mod: MOD_i)
(env: ENV_i)
(post: $POST_i$)).

requirements: All sd_i must be true in the current state.

effects: Adds the following state delta to the current state:

(SD (pre: (OR $case_1$ ... $case_n$)
pred)
(mod: MOD_1 ∪ ... ∪ MOD_n)
(env: ENV_1 ∪ ... ∪ ENV_n)
(post: (OR $POST_1$ ... $POST_n$))

(PerformInduction loop-sd base-sd)

meaning: Derives a state delta representing the state transformation from the start of a loop to its termination (the number of times through the loop being known in advance).

arguments: base-sd is a state delta representing the state transformation for the first time through the loop, and loop-sd is the state delta representing the state transformation once through the loop, starting after an arbitrary number of iterations. In the following *from* and *to* are numbers, *indvar* is a bitstring term, *claim*

is what is to be proved (written as a list (or conjunction) of predicates in the state delta expression language), and *program* is a state delta encoding the execution of the loop.

base-sd must be of the form:

```
(SD (pre:)
    (mod:)
    (env: OMEGA)
    (post: indvar=from
       claim(from/to)
           program)
```

loop-sd must be of the form:

```
(SD (pre: from ≤ indvar
          indvar ≤ to
          claim
          program)
    (mod: [no restriction])
    (env: )
    (post: indvar(#/.) = indvar + 1
           claim(#/.)
           program))
```

requirements: base-sd and loop-sd must be in the current state.

effects: If base-sd and loop-sd are in the current state, PerformInduction adds the following state delta to the current state:

```
(SD (pre: program)
    (mod: loop-sd:mod)
    (env: OMEGA)
    (post: indvar(#/.) = to
           claim(#/., to/indvar)
           program))
```

A.2.6 Mapping Between Levels

(ApplyMapping)

meaning: Searches the current state for an "applicable" mapping and "applies" it.

arguments: none

requirements: There must be an applicable mapping.

effects: Finds a mapping with mapping:from true in the current state, and adds mapping:to and mapping:map to the current state.

A.2.6 Static Reasoning

(InstantiateContents place var)

meaning: Instantiates the contents of place to be var.

arguments: Place is already registered and var is new; both are atoms.

requirements: Place must be registered, var must be new, and both must be atoms.

effects: Substitutes var for (.place) everywhere in the current state, and adds the predicate (.place)=var.

(Derive exp)

meaning: Inserts exp into the current state.

arguments: Typically exp is a predicate.

requirements: none

effects: Allows direct user alteration of the current state; thus would not be used in a completely system-checked proof.

A.3 HIGH LEVEL PROOFSTEPS

Our experience with detailed proofs has shown that there are patterns of proofstep sequences that can be lumped together to a single (more abstract) proofstep. High level proofsteps are generally only necessary for setting up a proof, for symbolic execution of straight line code, for execution of alternation, for execution of iteration, and for performing symbolic simulation.

The set of high level proofsteps forms a language that is compact and structured. Using this language makes it easier to read or write proofs.

(Prove sd proof)

meaning: Proves sd by proof.

arguments: sd is a state delta and proof is a list of proofsteps.

requirements: Those of Open.

effects: Performs (Open NIL sd) and then sequentially processes the elements of proof.

(ProposeMode breakpoint)

meaning: Symbolically executes from the current state until breakpoint is reached or until a (Close) can be performed.

arguments: breakpoint is a predicate.

requirements: none

effects: Checks to see if Breakpoint is true in the current state; if yes, halts; if not, checks to see if (Close) is possible; if yes, (Close) is performed; if not, checks to see if there is an applicable state delta sd; if yes, performs (ApplySD sd); if not, halts with the message "Proofchecker has nothing to propose".

(ProvebyCases sd case-proof-list)

meaning: Proves (a state delta equivalent to) sd, by using the case analysis specified in case-proof-list.

arguments: sd is a state delta, and case-proof-list is a list of the form

$$((case_1\ proof_1)\ \ldots\ (case_n\ proof_n))$$

where the cases are predicates specifying the different cases and the proofs are lists of proofsteps which prove sd in case $case_i$ is true.

requirements: Those of (CombineCases).

effects: Sequentially treats the elements of case-proof-list by adding pred to sd:pre and then sequentially processing proof. After the last element of case-proof-list is processed, (CombineCases (sd_1 ... sd_n)) is performed where sd_i is sd with $case_i$ added to its precondition.

(SymSimulate)

meaning: Proves a series of simulation relationships.

arguments: none

requirements: none

effects: Assumes that the goal is a list of state deltas to be proved (sd ...). For each sd in the goal performs the following sequence of proofsteps: (Open NIL sd), (ApplyMapping), (ProposeMode b), (ApplyMapping), (Close). The breakpoint b in ProposeMode is mapping:from of the mapping for which mapping:to is true in sd:post.

(InitProof program) meaning: Initializes the system in order to prove something (to be specified in a later (Prove) proofstep) about program.

arguments: program.isp is a file containing an ISPS program.

requirements: program must be a valid ISPS program.

effects: Translates program into the internal state delta representation, and initializes the placesystem using the information on the declared places in program.

A.4 STATE DELTA EXPRESSION LANGUAGE

In this section we describe the function symbols used in the state delta language. This language is intended to accommodate all the needs of the whole system, from translating a machine-description program in ISPS, to writing down the high level specification, to writing down the proof. Thus we deal with placenames (program identifiers), bitstrings, arrays, and several varieties of numbers.

DATA DOMAINS

P Places (in a machine; or in general any set of "names")

B Bitstrings

N Natural Numbers

Z Integers

Q Rationals

A Arrays (considered as a superset of B)

{T,NIL} Truth values

In the following we give the definitions of the function symbols. The constant bitstrings are value-length pairs written m(n) where $m<2^n$. Note that there is only one legal bitstring of length 0, that of value 0. The symbols =,+,-,*,and ≤ are logical equality, and arithmetical symbols. Additional "support functions" are mod, int(x)=integral part of x,

maxlh(a,b) = max{(LH a), (LH b)}, and tctous(i,n) (2's complement to unsigned), which takes $i \in Z$ and $n \in N$ such that $-2^{n-1} \leq i < 2^{n-1}$ and returns that non-negative number which is the unsigned value of the bitstring of length n representing i in 2's complement. Thus, tctous(i,n)=if $i \geq 0$ then i else 2^n+i. So, tctous(-3,4)=13, tctous(-4,3)=4, and tctous(-3,2) is undefined. Notice that in all the uses of tctous below, the arguments satisfy the conditions for the definition. "Exp=If p then x else y" is a short form of writing a definition of Exp by cases: If p is true, then Exp=x; if p is false, then Exp=y. The union of two sets is denoted by U; thus, for example, in the specification of LH, LH:PUAUN-->N means that LH is a function taking either a place, array (and hence bitstring), or number, and returning a number.

(DOT p) .p Contents of p

DOT:P-->A
DOT is an arbitrary function subject to the restrictions that (LH p)=(LH .p) and (HT p)=(HT .p).

(LH x) Length of x

LH:PUAUN-->N
The length of a place is an arbitrary natural number.
The length of an array is the same as the length of all its rows.
The length of a bitstring b is a natural number j such that $i < 2^j$, where i=(USVAL b).
The length of a natural number is one more than the number of binary digits needed to represent it.

(HT x) Height of x

HT:PUAUN--> N
The height of a place is any natural number
The height of an array is the number of its rows.
The height of a bitstring or natural number is 1.

(USVAL b) Unsigned value of bitstring b

USVAL:B-->N
The case by case definition is given below.
Note that Places do not have USVAL's; however Numbers, considered as bitstrings, do.

(TCVAL b) Two's complement value of b

TCVAL:B-->Z

(TCVAL b)=if (USVAL b)<$2^{(LH\ b)-1}$ then (USVAL b) else (USVAL b)-$2^{(LH\ b)}$

(VarBS i j) Bitstring of USVAL i (almost) and LH j

VarBS:NXN-->B

(USVAL (VarBS i j))=i mod 2^{j}

(LH (VarBS i j))=j

(BSEQL a b) Equality between bitstrings

BSEQL:BXB-->B

(BSEQL a b)=if (USVAL a)=(USVAL b) and (LH a)=(LH b) then 1(1) else 0(1)

(USCONC a b) Concatenation of a and b

USCONC:BXB-->B

(USCONC a b)=(VarBS [(USVAL a)*$2^{(LH\ b)}$ + (USVAL b)] (LH a)+(LH b))

(USSUB a m n) Substring of b from bit m down to n

USSUB:BXNXN-->B

(USSUB a m n)= if m>(LH a) then (USSUB a (LH a)-1 n)

elseif m<n then 0(0)

else (VarBS int(((USVAL a) mod 2^{m+1})*2^{-n}) m-n+1).

(USSUB a m) m-th bit of a

(USSUB a m)=(USSUB a m m)

(BITS p (PAIR m n)) Subplace of p from bit m down to n

BITS: PXNXN-->P

(DOT (BITS p (PAIR m n))=(USSUB (DOT p) m n)

(BITS p m) Alternative form for (BITS p (PAIR m m))

(DOT (BITS p m)=(USSUB (DOT p) m)

(BITPLUS a b) Same length bit addition

BITPLUS:BXB-->B

(BITPLUS a b)=(VarBS [(USVAL a)+(USVAL b) mod $2^{maxlh(a,b)}$] maxlh(a,b))

BITPLUS (essentially) zero-extends a and b to be the same length, adds them, and drops the carry, if any.

BITPLUS can be used to uniformly define USPLUS and TCPLUS.

(USPLUS a b) Unsigned addition

USPLUS:BXB-->B
(USPLUS a b)=(VarBS (USVAL a)+(USVAL b) maxlh(a,b)+1)
or:
(USPLUS a b)=(BITPLUS (USCONC (VarBS 0 maxlh(a,b)+1-(LH a)) a)
(USCONC (VarBS 0 maxlh(a,b)+1-(LH b)) b))

(TCPLUS a b) Two's complement addition

TCPLUS:BXB-->B
(TCPLUS a b) is that bitstring of length maxlh(a,b)+1 whose TCVAL is (TCVAL a)+(TCVAL b). There are several possible ways to describe that in terms of VarBS.
(TCPLUS a b)=
(VarBS tctous((TCVAL a)+(TCVAL b),maxlh(a,b)+1) maxlh(a,b)+1)).
Or in terms of BITPLUS:
(TCPLUS a b)=(BITPLUS (USCONC 0(1) (SE a maxlh(a,b)))
(USCONC 0(1) (SE b maxlh(a,b)))),
where SE is defined below.

(USDIFFERENCE a b) Unsigned difference

USDIFFERENCE:BXB-->B
(USDIFFERENCE a b)=
(VarBS tctous((USVAL a)-(USVAL b),maxlh(a,b)+1) maxlh(a,b)+1)

(TCDIFFERENCE a b) Two's complement difference

TCDIFFERENCE:BXB-->B
(TCDIFFERENCE a b)=
(VarBS tctous((TCVAL a)-(TCVAL b),maxlh(a,b)+1) maxlh(a,b)+1)

(USTIMES a b) Unsigned multiplication

USTIMES:BXB-->B
(USTIMES a b)=(VarBS (USVAL a)*(USVAL b) (LH a)+(LH b))

(TCTIMES a b) Two's complement multiplication

TCTIMES:BXB-->B
(TCTIMES a b)=
(VarBS tctous((TCVAL a)*(TCVAL b),(LH a)+(LH b)) (LH a)+(LH b)))

(USEQL a b) Unsigned equality

USEQL:BXB-->B
(USEQL a b)= if (USVAL a)=USVAL b) then 1(1) else 0(1)

(TCEQL a b) Two's complement equality

TCEQL:BXB-->B
(TCEQL a b)= if (TCVAL a)=(TCVAL b) then 1(1) else 0(1)

(USNEQ a b) Unsigned inequality

USNEQ:BXB-->B
(USNEQ a b)= if (USVAL a)≠(USVAL b) then 0(1) else 1(1)

and similarly for the other bit relations: TCNEQ, USLSS, TCLSS, USLEQ, TCLEQ, USGTR, TCGTR, USGEQ, TCGEQ

(BITMINUS a) Same length two's complement negation

BITMINUS:B-->B
(BITMINUS a)=(VarBS ($2^{(LH\ a)}$-(USVAL a) mod $2^{(LH\ a)}$) (LH a))

(USMINUS a) Unsigned negation

USMINUS:B-->B
(USMINUS a)=(VarBS tctous(-(USVAL a),(LH a)+1) (LH a)+1)

(TCMINUS a) Two's complement negation

TCMINUS:B-->B
(TCMINUS a)=(VarBS tctous(-(TCVAL a),(LH a)+1) (LH a)+1)

(SE a m) Sign extend a to length m

SE:BXN-->B
(SE a m) has the sign TCVAL as a (if m≥(LH a)). Thus:
(SE a m)= if m<(LH a) then (USSUB a m-1 0)
else (VarBS tctous((TCVAL a),m) m).

(USSLO a m) Shift left m bits shifting in 0

USSLO:BXZ-->B
(USSLO a m)= if m<0 then (USSRO a -m)
else (USCONC (USSUB a (LH a)-1-m 0) (USSUB (VarBS 0 (LH a)) m-1 0)).
This last clause can also be written as:
(VarBS (USVAL a)*2^{m} mod $2^{(LH\ a)}$ m)

(USSL1 a m) Shift left m bits shifting in 1

```
USSL1:BXZ-->B
(USSL1 a m)= if m<0 then (USSR1 a -m)
      else (USCONC (USSUB a (LH a)-1-m 0)
            (USSUB (VarBS 2^(LH a)-1 (LH a)) m-1 0))
```

(USSLR a m) Shift left rotate

```
USSLR:BXZ-->B
(USSLR a m)= if m<0 then (USSRR a -m) else
      (USCONC (USSUB a (LH a)-m-1 0) (USSUB a (LH a)-m))
```

(USSLD a m) Shift left duplicate right bit

```
USSLD:BXZ-->B
(USSLD a m)= if (USVAL (USSUB a 0 0))=1 then (USSL1 a m)
      else (USSL0 a m)
```

(USSR0 a m) Shift right m shifting in 0

```
USSR0:BXZ-->B
(USSR0 a m)= if m<0 then (USSL0 a -m)
      else (USCONC (USSUB (VarBS 0 (LH a)) m-1 0) (USSUB a (LH a)-1 m))
```

(USSR1 a m) Shift right m shifting in 1

```
USSR1:BXZ-->B
(USSR1 a m)= if m<0 then (USSL1 a -m)
      else (USCONC (USSUB (VarBS 2^(LH a)-1 (LH a)) m-1 0)
            (USSUB a (LH a)-1 m))
```

(USSRR a m) Shift right rotate

```
USSRR:BXZ-->B
(USSRR a m)= if m<0 then (USSLR a -m)
      else (USCONC (USSUB a (LH a)-1 m) (USSUB a m-1 0))
```

(USSRD a m) Shift right duplicate left bit

```
USSRD:BXZ-->B
(USSRD a m)=if (USVAL (USSUB a (LH a)-1 (LH a)-1))=1 then (USSR1 a m)
      else (USSR0 a m)
```

Note that all of the results of the shifts have length (LH a)

(USNOT a) Bitstring-logical NOT

USNOT:B-->B

(USOR a b) Bitstring-logical OR

USOR:BXB-->B
Zero-extends to maximum length and ORs

(USAND a b) Bitstring-logical AND
(USEQV a b) Bitstring-logical equivalence
(USXOR a b) Bitstring-logical exclusive OR
Similarly

(EXPVAL a) TCVAL of right 8 bits

EXPVAL:B-->Z
(EXPVAL a)=(TCVAL (USSUB a 7 0))

(MANVAL a) Fractional value of left 24 bits

MANVAL:B-->Q
(MANVAL a)=(TCVAL (USSUB a 31 8))* 2^{-23}

(FLVAL a) Value of a as a floating number

FLVAL:B-->Q
(FLVAL a)=(MANVAL a)*$2^{(EXPVAL\ a)}$

(NZEROP a) Not zero predicate

NZEROP:B-->{T,NIL}
(NZEROP a)= if (USVAL a)=0 then NIL else T

(POWER q i) Integer exponentiation of rationals

POWER:QXZ-->Q

(REALMINUS q) Unary arithmetic negation

REALMINUS:Q-->Q

(PRODUCT q r) Multiplication
(REALPLUS q r) Addition
(REALDIFFERENCE q r) Subtraction
(REALQUOTIENT q r) Division

All these from QXQ-->Q

(REALEQUAL q r) (Provable) equality between arithmetic terms

REALEQUAL:QXQ-->B
(REALEQUAL q r)=if q=r then 1(1) else 0(1)

(REALLEQ q r) (Provable) less than or equality

REALLEQ:QXQ-->B
(REALLEQ q r)= if q≤r then 1(1) else 0(1)

Now we describe the terms dealing with arrays. Two arrays are the same iff they have the same height and the same sequence of words. Thus: We have no function analogous to USVAL for arrays, although it is an easy matter to uniquely assign a number to an array on the basis of the USVALs of its words. We number the rows of an array from top to bottom, starting with 0. We have learned to view as natural the apparent discrepancy between the top-down ordering of rows in an array and the right-left ordering of bits in a bitstring.

(WORDS a m n) The rows of a from m down to n

WORDS:AXNXN-->A
(HT (WORDS a m n))=if n≥(HT a) then (HT (WORDS a m (HT a)-1))
elseif m>n then 0
else n-m+1

(WORDS a m) m-th word of a

(WORDS a n)=(WORDS a n n)

(SUBARRAY a i j) The columns of a from i to j

SUBARRAY:AXNXN-->A
(HT (SUBARRAY a i j))=(HT a)
(WORDS (SUBARRAY a i j) m m)=(USSUB (WORDS a m m) i j)

(RANGE a) The concatenation of the rows of a

RANGE:A-->B
(RANGE a)=(USCONC (WORDS a 0 0)...(WORDS a (HT a)-1 (HT a)-1))
It is convenient to define (RANGE x y) for two bits of the explicit form
x=(USSUB (WORDS a jx jx) ix ix) and y=(USSUB (WORDS a jy jy) iy iy)
or in the degenerate case where a is of length 1,
x=(WORDS a jx jx) and y=(WORDS a jy jy). Its value is the word
consisting of all bits from x to and including y inside a.

(ARRAYNGE h b) Forms b into an array of height h .

ARRAYNGE:NXB-->A
Defined only for b such that h|(LH b)
(HT (ARRAYNGE h b))=h
(WORDS (ARRAYNGE h b) i i)= if i<h then
(USSUB b (LH b)-1-i*(LH b)/h)
else 0(0)

(ARRAYCONC h a b) Forms a and b into an array of height h

ARRAYCONC:NXAXA-->A
Defined only for h,a,b such that h divides the areas of a and b.
(HT (ARRAYCONC h a b))=h
(WORDS (ARRAYCONC h a b) i i)= if i<h then
(USCONC (WORDS (ARRAYNGE h (RANGE (USSUB (WORDS a 0 0) (LH a)-1)
(USSUB (WORDS a (HT a)-1 (HT a)-1) 0 0))) i i)
(WORDS (ARRAYNGE h (RANGE (USSUB (WORDS b 0 0) (LH b)-1)
(USSUB (WORDS b (HT b)-1 (HT b)-1) 0 0))) i i))
else 0(0)

A.5 THE SIMPLIFIER

SIMPLIFIER STRUCTURE

In this section we describe the structure of the simplifier and give a brief description of the purpose of each of its files. Entry to the simplifier is through the function *SIMPEVAL*. *SIMPEVAL*(X) returns a term equivalent to X if X is a term (legal expression) in the simplifier's language. The simplification is processed recursively; that is, if X is not atomic, then the arguments of X are first passed to *SIMPEVAL*, and likewise for their arguments. If no simplifcation or evaluation is possible (by the system) then the original argument is returned.

SIMPLIFY

SIMPLIFY consists of two levels. At the top level, the function *SIMPEVAL* is the entry point to the simplifier. An expression to be simplified is sent to the appropriate second level routine by *SIMPEVAL* after its arguments have been recursively simplified by the same process. This appropriate routine is chosen on a one-to-one basis depending on the principal function symbol of the expression.

The second level routines consist of three parts: if the simplified arguments are not symbolic, the expression is evaluated and the value returned;[4] if not, then the expression is passed to one of the files listed below for further processing; if this does not result in further simplification, the original expression with simplified arguments is returned.

If the expression is of type real numbers or integers, or relations on them, and the simplified arguments are constant numbers, then the evaluation is done by LISP functions. If the arguments are symbolic, then the computation calls a routine in REALSIMP.

If the expression is of type bitstring and the arguments are constant bitstrings, then the evaluation is done by functions in MDTE. If the arguments are symbolic then the computation calls a routine in ISPSSIMP.

If the expression is of type value of bitstring, and the arguments are constant bitstrings, then the evaluation is done in SIMPLIFY by LISP functions and perhaps other second level functions. If the arguments are symbolic, the computation calls a routine in VALUESIMP.

If the expression is of type arrays then ARRAYSIMP is called.

If the expression is of type propositional calculus, and the arguments are not logical constants (T or NIL), then LOGSIMP is called.

---

[4] This convention is not strictly observed; some functions at this level do simplification on symbolic expressions and/or examine the data base.

Each of these files may call SIMPLIFY, each other, or OTHERBITSIMP and AUXILIARYSIMP. In addition, they all search the data base for current facts which may imply some simplification that is not generally true.

REALSIMP

This file contains the main routines for simplification of algebraic expressions over the domain of the real numbers. The relations and functions recognized, along with their internal syntax, are addition (REALPLUS), subtraction (REALDIFFERENCE), multiplication (PRODUCT), division (REALQUOTIENT), exponentiation (POWER), unary negation (REALMINUS), equality (REALEQUAL), strict order (REALLESS), and weak order (REALLEQ).

In addition, the maximum and/or minimum bound on a real variable is found where possible by searching the data base for the entries of the form (REALLEQ var n) or (REALLEQ n var) where n is a numerical constant. The internal syntax for these minimum and maximum values is REALMIN and REALMAX.

ISPSSIMP

ISPSSIMP is the file simplifying bitstring expressions (more or less those of ISPS). An important point is that we allow bitstring variables to have variable lengths (including zero) as well as variable contents. A constructor expression (formed of concatenation, substring selection, and shifts) is reduced to a standard form as a concatenation of substrings, where two adjacent substrings may not be combined any further. This standard expression is almost canonical; that is, two equivalent bitstrings reduce to the same standard expression except in certain cases involving registers whose variable length may include zero.

Two's complement or unsigned plus and difference are replaced by an equivalent addition or subtraction between two bitstrings of equal length and sent to OTHERBITSIMP for processing. In the case of bitstring multiplication, some simplification is accomplished if one of the arguments is a bitstring with known value.

If the expression is an equality between bitstrings, then simplification is accomplished in many cases, either completely (i.e., to the bitstrings 1 or 0 representing T and F) or partially. There is also some use made of REALSIMP and VALUESIMP, for example, in the

equivalence between unsigned equality of bitstrings and real equality between their unsigned values.

OTHERBITSIMP

OTHERBITSIMP contains routines for use in simplifying bitstring expressions, and is in principle subordinate to ISPSSIMP. Included are routines for simplifying the non-carry bitstring addition BITPLUS, sign-extension, substrings of concatenations, squashing together two adjacent substrings in a concatenation, and replacing a substring of the form A<lh(a)-1:0> by A.

VALUESIMP

The two main expressions simplified in VALUESIMP are USVAL(A) and TCVAL(A), the unsigned and two's complement value of the bitstring A. In addition FLVAL(A), EXPVAL(A), and MANVAL(A) are expressions representing the value of A as a floating number (of customized 24-bit mantissa and 8-bit exponent), the two's complement value of the exponent of A, and the two's complement value of the mantissa of A, respectively.

Typical steps in a recursive simplification are changing a TCVAL into a USVAL where possible (and sending the result back to *SIMPEVAL*), changing TCVAL(A) into TCVAL(B) where B is simpler than A, returning an integer instead of TCVAL or USVAL, or "pushing TCVAL in" and returing an expression of the form TCVAL(A)+TCVAL(B).

ARRAYSIMP

ARRAYSIMP simplifies expressions in the array language described in Microver Note #12. This language allows all possible row and column and subarray selection, reshaping, and concatenation of two rectangular arrays of constant height and length. It is completely integrated with the bitstring language in that a word in an array is a bitstring, an array of height 1 is a word, and the length of an array is the (common) length of its words. The height and area of arrays are calculated here, but the length is calculated in AUXILIARYSIMP.

LOGSIMP

LOGSIMP recognizes formulas of the propositional calculus written with implication and disjunction. Free individual variables are allowed, and in this case we treat the formula

as if all the free variables were universally quantified.

AUXILIARYSIMP

This file contains the simplifications of the other "service" functions used in the simplifier. First, we have the representation of an arbitrary continuous piecewise linear function on bounded domain:

$$(\text{SLANT } v\ (a\ h)\ (l_1\ s_1)\ (l_2\ s_2)\ \ldots\ (l_n\ s_n)).$$

where v is the function's argument variable, a is the left endpoint, h is the height of the graph at a, and from then on the graph continues l_1 units to the right with slope s_1, and then l_2 units with slope s_2, etc. There are routines for adding slant functions, finding maximum or minimum of two slants, converting from standard arithmetic notation to slant notation, etc. Slants are used mostly as lengths of variable length bitstrings.

There are routines for calculating the length of bitstring expressions, inserting and extracting parentheses, "multiplying out" arithmetic expressions, solving linear equations, and converting from rationals to bitstrings representing them in floating point format.

PRINCIPLES

In the following we describe the principles behind some simplifications for expressions in the state delta language. This is not intended to be a complete survey of all possible simplifications, but rather a representative list of those simplifications found useful in the actual practice of verification, especially the square root algorithm of the FTSC. Thus there is a close correspondence between these simplifications and those actually implemented in the system. Here, though, we describe only the "interesting" ones, and some of these may be stated in different form without mentioning all the cases and specifying the implementation details.

BSC (bitstring constructor) terms

The primitive operations for constructing bitstrings are concatenation a@b, substring selector a<i:j>, and shifts. The definitions of concatenation and shifts are standard. Our conventions for substring selector are that bitstrings are numbered from the right-most bit a<0> to the left-most a<lh(a)-1> where lh(a) is the length of a. Note that we shall

allow bitstrings to have variable length. These are called generalized bitstrings. For integer i, j a<i:j> represents the string consisting of bits i down to j of a, that is, a<i>@a<i-1>...@a<j>. If j is greater than i, then this string is nonexistent, and is called EMPTY. If i<0 or i≥lh(a) then a<i> is EMPTY. In the following f(i) and g(i) will be functions attaining integer values at integer values of the argument i. We will occasionally omit mention of i and write just f, g.

A (generalized) substring is a term of the form a<f:g> where a is atomic.

A simplified substring is the EMPTY string or is a substring of the form a<f:g> where

$\forall i\ f(i) < lh(a)$, $\forall i\ g(i) \geq 0$, $\neg\forall i\ f(i) < g(i)$.

Note that when f and g are constants, these conditions become f<lh(a), g≥0, f≥g. Note also that we cannot demand ∀i f(i)≥g(i), since for example a<0:-i> is either EMPTY or a<0> depending on i. From our definition of the semantics of substring, it follows that any substring is equivalent to a simplified substring: a<f:g>= a<min{f, lh(a)-1}, max{g,0}> or EMPTY. If a canonical simplified substring is desired, some standard values of f and g will have to be taken in the case that f(i)<g(i), for example f(i)=0 and g(i)=1.

Length is defined for a (generalized) substring as the following function of i: (Let a, f, and g be functions of i)

```
lh(a<f:g>)(i) = if f(i)≥ lh(a(i)) then lh(a<lh(a)-1:g>(i))
                elseif g(i) < 0 then lh(a<f:0>(i))
                elseif f(i)<g(i) then 0
                else f(i)-g(i)+1.
```

An equivalent closed form is

$$lh(a\langle f:g\rangle) = \min\{lh(a), \max\{\min\{f, lh(a)-1\} - \max\{g, 0\} + 1, 0\}\}$$

This allows the following rewriting: Let 0(1) denote a string of f zeroes.

. If a is of the form 0(f)<g:h>, then a ⇒ 0(lh(a)). (1)

A BSC (bitstring constructor) term is any term formed from atomic bitstrings, concatenation, substring, and shifts.

A simplified BSC term is of the form $b_1@b_2@...@b_n$ where $n \geq 1$ and each b_i is a simplified substring.

It can be shown that every BSC term is equivalent to a simplified BSC term. The main simplification rules used in simplifying a BSC term are

$$(a@b)\langle f:g\rangle \Rightarrow a\langle f-lh(b): g-lh(b)\rangle \ @ \ b\langle f:g\rangle \tag{2}$$

$$a \text{ SLO } f \Rightarrow 0(\min\{lh(a),-f\})@a\langle lh(a)-f-1:\max\{-f,0\}\rangle@0(\min\{lh(a),f\}) \tag{3}$$

$$a\langle f_1:g_1\rangle\langle f_2:g_2\rangle \Rightarrow a\langle \min\{f_1,f_2+g_1\}:\max\{g_1,g_1+g_2\}\rangle \tag{4}$$

Example Assume lh(a)=4, lh(b)=5, lh(c)=6.

$(a@(b@c) \text{ SLO } 5)\langle 13:3\rangle\langle 6:1\rangle \Rightarrow$

$(0(-5)@(a@(b@c))\langle 9:0\rangle@0(5))\langle 9:4\rangle \Rightarrow$

$(EMPTY@(a\langle -2:-11\rangle@(b@c)\langle 9:0\rangle)@0(5))\langle 9:4\rangle \Rightarrow$

$(b\langle 3:0\rangle@c\langle 9:0\rangle@0(5))\langle 9:4\rangle \Rightarrow$

$(b\langle 3:0\rangle@c@0(5))\langle 9:4\rangle \Rightarrow$

$c\langle 4:0\rangle@0(1)$

BSA (bitstring arithmetic) terms

All the bitstring addition operators are translated into BITPLUS; BITPLUS is noncarry addition between two bitstrings of equal length. When the sign + appears between bitstrings it will always denote BITPLUS. We also use + for numerical addition, but it is clear from the context which is intended. USVAL(a) is the nonnegative integer represented in binary by the bitstring a.

If b and c are constant bitstrings and $USVAL(b)+USVAL(c) < 2^{lh(b)}$, then

$$(a@b)+c \Rightarrow a@(b+c)\langle lh(b)-1:0\rangle \tag{5}$$

A similar simplification rule holds for c+(a@b). Of course the two sides of 5 are equivalent even if b and c are not constants, but then the right side is not necessarily

simpler.

BSR (bitstring relational) terms

There are two main classes of bitstring relations: unsigned value and two's complement. Every unsigned bitstring relation is equivalent to the the corresponding real relation on the USVAL's of its arguments. For example, USEQL(a,b) is equivalent to USVAL(a)=USVAL(b). Similarly for two's complement. The simplification of this type of relation will be given in this section. The section on real relations will include (among others) "mixed relations", i.e., those containing both USVAL and TCVAL. TCVAL(a) is the (signed) integer which is the two's complement interpretation of the bitstring a.

Equality

We let $a =_{US} b$ denote USEQL(a,b)=T and similarly for TCEQL. We write = with no subscript if identity between bitstrings is intended.

If $\forall ij\ (f_1(i)<j\leq f_2(i) \vee f_2(i)<j\leq f_1(i) \rightarrow a\langle j\rangle=0)$, then

$$a\langle f_1:g\rangle =_{US} a\langle f_2:g\rangle \tag{6}$$

If $a_1 =_{US} a_2$ and $b_1 =_{US} b_2$ and $lh(b_1)=lh(b_2)$,or if $b_1 =_{US} b_2\langle lh(b_1)-1:0\rangle$ and $a_1 =_{US} a_2@b_2\langle lh(b_2)-1:lh(b_1)\rangle$, then

$$a_1@b_1 =_{US} a_2@b_2 \tag{7}$$

If $a =_{US} 0$ and $b =_{US} 0$, then

$$a@b =_{US} 0 \tag{8}$$

Of course, there are the obvious generalizations when an arbitrary constant is in place of 0.

If $a_1 =_{US} a_2$ and $b_1 =_{US} b_2$ or $a_1 =_{US} b_2$ and $b_1 =_{US} a_2$, then

$$a_1+b_1 =_{US} a_2+b_2 \tag{9}$$

If $USVAL(a)\geq 2^{lh(a)}-2^f$ or $0>TCVAL(a)>-2^f-1$, then

$$a\langle f\rangle = 1 \tag{10}$$

If $a\langle f_1:g_1\rangle =_{US} 0$ for some $f_1\geq f$, $g_1\leq g$, then

$$a\langle f{:}g\rangle =_{US} 0 \tag{11}$$

If $a =_{US} b$ and $a\langle lh(a)-1\rangle = b\langle lh(b)-1\rangle$ (or $lh(a)=lh(b)$), then

$$a =_{TC} b \tag{12}$$

If $a\langle f\rangle = a\langle f+1\rangle = \ldots = a\langle lh(a)-1\rangle$, then

$$a\langle f{:}0\rangle =_{TC} a \tag{13}$$

If $a\langle f+1\rangle = a\langle f\rangle = a\langle f-1\rangle$ and $b\langle f+1\rangle = b\langle f\rangle = b\langle f-1\rangle$, then

$$(a \pm b)\langle f\rangle = (a \pm b)\langle f+1\rangle \tag{14}$$

If $f_1-g_1=f_2-g_2$, $a\langle f_1'{:}g_1'\rangle =_{US} b\langle f_2'{:}g_2'\rangle$, $f_1'\geq f_1$, $g_1'\leq g_1$, $f_1'-f_1 = f_2'-f_2$, $g_1-g_1' = g_2-g_2'$;

or if $a\langle lh(a)-1{:}g_1\rangle =_{US} b\langle lh(b)-1{:}g_2\rangle$, $a\langle f_1+1\rangle = \ldots = a\langle lh(a)-1\rangle = 0$, $b\langle f_2+1\rangle = \ldots = b\langle lh(b)-1\rangle = 0$,

then

$$a\langle f_1{:}g_1\rangle =_{US} b\langle f_2{:}g_2\rangle \tag{15}$$

Ordering

$$0 \leq_{TC} a \tag{16}$$

if and only if $a\langle lh(a)-1\rangle = 0$.

BSV (bitstring value) terms

If $a\langle lh(a)-1\rangle = 0$, then

$$TCVAL(a) \Rightarrow USVAL(a) \tag{17}$$

If $a\langle lh(a)-1\rangle = 0$, then

$$USVAL(a) \Rightarrow USVAL(a\langle lh(a)-2{:}0\rangle) \tag{18}$$

$$TCVAL(a@b) \Rightarrow 2^{lh(b)}*TCVAL(a) + USVAL(b) \tag{19}$$

$$USVAL(a@b) \Rightarrow 2^{lh(b)}*USVAL(a) + USVAL(b) \tag{20}$$

If $lh(a)=lh(b)$, $a\langle f-1\rangle = b\langle f-1\rangle = 0$, $a\langle f\rangle = a\langle f+1\rangle = \ldots = a\langle lh(a)-1\rangle$, $b\langle f\rangle = b\langle f+1\rangle = \ldots = b\langle lh(b)-1\rangle$, then

$$TCVAL((a+b)\langle f{:}0\rangle) \Rightarrow TCVAL(a+b) \tag{21}$$

If $lh(a)=lh(b)$ and $TCVAL(a) + TCVAL(b) \geq 2^{lh(a)-1}$, then

$$TCVAL(a+b) \Rightarrow TCVAL(a)+TCVAL(b)-2^{lh(a)} \tag{22}$$

If $lh(a)=lh(b)$ and $TCVAL(a) + TCVAL(b) < -2^{lh(a)-1}$, then

$$TCVAL(a*b) \Rightarrow TCVAL(a) + TCVAL(b) + 2^{lh(a)} \tag{23}$$

If $lh(a)=lh(b)$ and $-2^{lh(a)-1} \leq TCVAL(a) + TCVAL(b) < 2^{lh(a)-1}$, then

$$TCVAL(a*b) \Rightarrow TCVAL(a) + TCVAL(b). \tag{24}$$

RA (real arithmetic) terms

We list here only the rules concerning RA terms which contain BSV terms.

Let c_1 and c_2 be functions of i (as are the f's and g's). If $c_1,c_2>0$, $f_1 \geq f_2$, $g_1=g_2$, and $\forall i(c_1(i) \neq c_2(i) \Rightarrow g_2(i)>f_2(i))$, then

$$c_1 * v(a\langle f_1:g_1\rangle) - c_2 * v(a\langle f_2:g_2\rangle) \Rightarrow \tag{25}$$
$$c_1 * 2^{max(f_2-g_2+1,0)} * v(a\langle f_1:g_1+max(f_2-g_2+1,0)\rangle).$$

Note that we do not demand that $\forall i(f_2 \geq g_2)$.

If $a\langle lh(a)-1\rangle=1$, then

$$TCVAL(a) + 2^{lh(a)} \Rightarrow USVAL(a). \tag{26}$$

RR (real relational) terms

$$TCVAL(a\langle lh(a)-1:n\rangle) \leq 2^{-n} * TCVAL(a) \tag{27}$$

Appendix B
FTSC HOST

FTSC HOST

```
!FTSC.MICROMACHINE

!This version (Mar.24, 1978) has made it through slftst 618-69A(16).

MICROFTSC:=(

**Main.Memory**

MEM[0:32K]<31:0>!ACTUALLY MEM IS 40 BITS WIDE BUT HERE
!WE JUST DEAL WITH THE PART THAT FITS
!ONTO THE CPU DATA BUS.

**ROM**

!FTSC.CONTROM          P214-216


CONTROM1[0:1023]<31:0>,!THREE SLICES OF CONTROM
CONTROM2[0:1023]<31:0>,
CONTROM3[0:1023]<13:0>,
MICWORD1<31:0>,
MICWORD2<31:0>,
MICWORD3<13:0>,
RF01<4:0>,    !MICWORD1<31:27>,
RF02<9:0>,    !MICWORD1<26:17>,
RF03<2:0>,    !MICWORD1<16:14>,
RF04<2:0>,    !MICWORD1<13:11>,
RF05<2:0>,    !MICWORD1<10:8>,
RF06<2:0>,    !MICWORD1<7:5>,
RF07<0>,    !MICWORD1<4>,
RF08<0>,    !MICWORD1<3>,
RF09<2:0>,    !MICWORD1<2:0>,
RF10<2:0>,    !MICWORD2<31:29>,
RF11<0>,    !MICWORD2<28> ,
RF12<0>,    !MICWORD2<27> ,
RF13<0>,    !MICWORD2<26> ,
RF14<0>,    !MICWORD2<25> ,
RF15<2:0>,    !MICWORD2<24:22> ,

RF16<2:0>,    !MICWORD2<21:19> ,
RF17<3:0>,    !MICWORD2<18:15> ,
RF18<3:0>,    !MICWORD2<14:11> ,
RF19<0>,    !MICWORD2<10> ,
RF20<0>,    !MICWORD2<9> ,
```

```
RF21<0>,    !MICWORD2<8> ,
RF22<2:0>,    !MICWORD2<7:5> ,
RF23<0>,    !MICWORD2<4> ,
RF24<0>,    !MICWORD2<3> ,
RF25<0>,    !MICWORD2<2> ,
RF26<0>,    !MICWORD2<1> ,
RF27<0>,    !MICWORD2<0> ,
RF28<0>,    !MICWORD3<13>,
RF29<0>,    !MICWORD3<12>,
RF30<0>,    !MICWORD3<11>,
RF31<4:0>,    !MICWORD3<10:6>,
RF32<0>,    !MICWORD3<5>,
RF33<0>,    !MICWORD3<4>,
RF34<0>,    !MICWORD3<3>,
RF35<0>,    !MICWORD3<2>,
RF36<0>,    !MICWORD3<1>,
RF37<0>,    !MICWORD3<0>,

!FTSC.ROMSEQUENCER       P213,217

RECONFIGROM[0:1023]<31:0>,!RECONFIGURATION ROM P121
!RECONFIGROM:=MEM["F7FF:"F000]  P69
RAD<9:0>,!NEXT ROM ADDRESS
ROMA4<0>:=RF02<5>,
ROMA5<0>:=RF02<4>,
ROMA6<0>:=RF02<3>,
ROMA7<0>:=RF02<2>,
ROMA8<0>:=RF02<1>,
ROMA9<0>:=RF02<0>,


AMODE<0>,!=0 IFF ADDRESS MODE=0
MONMD<0>,!=1 IN MONITOR CPU
CNTRL<0>,!=1 IF CONTROL PANEL WANTS ACCES TO CPU
SUMM1<0>,!"SUM<32>".THE INPUTS TO THE ALU
!ARE SIGN-EXTENDED TO 40 BITS AND THEN A DIFFERENCE BETWEEN
!SUMM1 AND SUM<31> INDICATES OVERFLOW (OVFF).
SUMM2<0>,!"SUM<33>"

!FTSC.ROMFUNCTIONDECODER       P.220

RFD00<0>,
RFD01<0>,
RFD02<0>,
RFD03<0>,
RFD04<0>,

RFD05<0>,
RFD06<0>,
RFD07<0>,
RFD08<0>,
RFD09<0>,
RFD10<0>,
```

```
RFD11<0>,
RFD12<0>,
RFD13<0>,
RFD14<0>,
RFD15<0>,
RFD16<0>,
RFD17<0>,
RFD18<0>,
RFD19<0>,
RFD20<0>,
RFD21<0>,
RFD22<0>,
RFD23<0>,
RFD24<0>,
RFD25<0>,
RFD26<0>,
RFD27<0>,
RFD28<0>,
RFD29<0>,
RFD30<0>,
RFD31<0>


**External.Connections**


SETROM:=SETROM(CONTROM1, CONTROM2, CONTROM3)

**Registers**

!FTSC.GENERALPURPOSEREGISTERS    P209

MANGPR[0:7]<23:0>,!8 MANTISSA GEN PURP REGS
MANGPRIN<23:0>,!FICTITIOUS MANTISSA INPUT
EXPGPR[0:7]<7:0>,!8 EXPONENT GEN PURP REGS
EXPGPRIN<7:0>,!FICTITIOUS EXPONENT INPUT

    !FTSC.WORKINGREGISTERS          P:209


MANWR[0:7]<23:0>,!8 MANTISSA WORKING REGISTERS
EXPWR[0:7]<7:0>,!8 EXPONENT WORKING REGISTERS
MANEXTREG<23:0>:=MANWR[4]<23:0>,!MANTISSA EXTENSION REGISTER
EXPEXTREG<7:0>:=EXPWR[4]<7:0>,!EXPONENT EXTENSION REGISTER
MANMEMDAT<23:0>:=MANWR[5]<23:0>,!MANTISSA MEMORY DATA
EXPMEMDAT<7:0>:=EXPWR[5]<7:0>,!EXPONENT MEMORY DATA
MANMEMADD<23:0>:=MANWR[6]<23:0>,!MANTISSA MEMORY ADDRESS
EXPMEMADD<7:0>:=EXPWR[6]<7:0>,!EXPONENT MEMORY ADDRESS
MANPC<23:0>:=MANWR[7]<23:0>,!MANTISSA PROGRAM COUNTER
EXPPC<7:0>:=EXPWR[7]<7:0>,!EXPONENT PROGRAM COUNTER
MANWRIN<23:0>,!FICTITIOUS MANTISSA INPUT
EXPWRIN<7:0>,!FICTITIOUS EXPONENT INPUT
MANWX<23:0>,!MANTISSA WX OUTPUT
```

```
WRYB<31:0>,!WRYB OUTPUT
MANWRYB<23:0>:=WRYB<31:8>,!MANTISSA WRYB OUTPUT
EXPWRYB<7:0>:=WRYB<7:0>,!EXPONENT WRYB OUTPUT
EXPWX<7:0>,!EXPONENT WX OUTPUT


!FTSC.INSTRUCTIONREGISTERP209,213

INR<31:0>,!INSTRUCTION REGISTER
RA<2:0>:=INR<21:19>,!SEE P60
RB<2:0>:=INR<24:22>,

!OTHER REGISTERS

HSW1<15:0>,!HARDWARE STATUS WORD 1
HSW2<31:0>,!HARDWARE STATUS WORD 2
MRAR<15:0>,!MOST RECENT ADDRESS REGISTER
MONMSKREG<31:0>,!MONITOR MASK REGISTER (REALLY?)



!FTSC.PIN    (PRIORITY INTERRUPT NETWORK) P229 FF

PERMSKREG<31:0>,!PERIPHERAL MASK REGISTER
INTREQREG<7:0>,!INTERRUPT REQUEST REGISTER
!HOW IS THIS LOADED? SEE 236 AND 112.
!RTI AND ARFLT ARE LOADED FROM INSIDE CPU.
!THE BITS CORRESPOND TO INTERRUPTS IN THE ORDER
!GIVEN ON P74 FOR INTMSKREG.
INTREQFF<7:0>,!INTERRUPT REQUEST FLIPFLOPS

REQPRIORITY<31:0>,!HIGHEST ON-BIT OF INTREQFF
INTMSKREG<7:0>,!INTERRUPT MASK REGISTER
PENDING<7:0>,!PENDING INTERRUPTS REGISTER
PRIORITYLEVEL<31:0>,!PENDING INTERRUPT PRIORITY LEVEL P112

INPROCFF<31:0>,!INTERRUPT IN-PROCESS FLIPFLOPS

ENADISFF<0>!ENABLE/DISABLE FLIPFLOP(1=DISABLE?)

☆☆ALU☆☆

!FTSC.ALUINPUTSELECTOR                P.208

!MANTISSA

MANINA<23:0>,!MANTISSA ALU A INPORT
MANA25<25:0>,!EXTENDED INPUT FOR
!CALCULATING OVERFLOW AND CARRY
MANINB<23:0>,!MANTISSA ALU B INPORT
MANB25<25:0>,

!EXPONENT
```

```
EXPINA<7:0>,!EXPONENT ALU A INPORT
EXPA9<8:0>,
EXPINB<7:0>,!EXPONENT ALU B INPORT
EXPB9<8:0>,

!FTSC.FUNCTIONINVERSION          P.207

!MANTISSA

MANCIN<0>,!MANTISSA CARRYIN
MANSELECT<3:0>,!MANTISSA S0-S3
INVMFN<0>,!MANTISSA INVERTER
EXPCOUT<0>,!EXPONENT ALU CARRY-OUT

!EXPONENT

EXPCIN<0>,!EXPONENT CARRY IN
EXPSELECT<3:0>,!EXPONENT S0-S3
INVEFN<0>,!EXPONENT INVERTER


!FTSC.ALUFUNCTIONSELECTOR          P.206

!MANTISSA  ALU OUTPUT FUNCTION  (IN2=0)

MANOVF<0>,!MANTISSA OVERFLOW
MANCOUT<0>,!MANTISSA CARRY OUT


!EXPONENT ALU OUTPUT FUNCTION


EXPOVF<0>,!EXPONENT OVERFLOW

!FTSC.AUTOMULDIVSUBP222

!AUTOMULTIPLY FUNCTION     P224

!THIS IS A VERY TENTATIVE VERSION

AUTOMULFN<3:0>,!AUTO MULTIPLY BITS
INVERTOR<0>:=AUTOMULFN<3>,!INVERT ALU FUNCTION
ALUBLS<0>:=AUTOMULFN<2>,!ALUB LEFT SHIFT
ALUBZ<0>:=AUTOMULFN<1>,!ALUB ZEROS
CRYSTS<0>:=AUTOMULFN<0>,!INTERNAL CARRY STATUS
MULBITS<1:0>,!MULTIPLIER BITS


!FTSC.ALUOUTPUTS             P204

SUM<31:0>,!SUM OUTPUT
```

```
ZDT8<0>,!ZERO DETECT SIGNALS
ZDT24<0>,
ZDT32<0>,
FDT32<0>,!FULL DETECT SIGNAL
CRYB00<0>,!MANTISSA CARRYOUT
OVF8<0>,!EXPONENT OVERFLOW
       !WHAT ABOUT MANTISSA OVERFLOW?

EXG23<0>!=1 IFF EXPONENT (I.E.SUM<7:0)> GTR 23


**Signals.Flipflops**

!FTSC.ENDCONDITIONSGENERATOR      P223



!NOTE THE QUESTION MARKS BELOW!
ENDCONDS<12:0>,!LIST OF ENDCONDITIONS
SUMMRS<0>:=ENDCONDS<12>,
SUMM2RS<0>:=ENDCONDS<11>,
SUMMLS<0>:=ENDCONDS<10>,
SUMM2LS<0>:=ENDCONDS<9>,
SUMELS<0>:=ENDCONDS<8>,
SUME2LS<0>:=ENDCONDS<7>,
WRYMRS<0>:=ENDCONDS<6>,
WRYM2RS<0>:=ENDCONDS<5>,
WRYMLS<0>:=ENDCONDS<4>,
WRYM2LS<0>:=ENDCONDS<3>,
WRYERS<0>:=ENDCONDS<2>,
WRYELS<0>:=ENDCONDS<1>,
WRYE2LS<0>:=ENDCONDS<0>,

!FTSC.FSFG(FLAG AND SPECIAL FUNCTION GENERATOR)
!P.226,94

EXTADD<0>,!EXTERNAL (TO THE CPU) ADDRESS SIGNAL
ROMADD<0>,!RECONFIGURATION ROM ADDRESS
HSW1EN<0>,!HARDWARE STATUS WORD 1 ENABLE
HSW2EN<0>,!HARDWARE STATUS WORD 2 ENABLE
LDBMRAR<0>,!LOAD MRAR; HOW IS THIS SET? SEE 229.
PERMSK<0>,!PERIPHERAL MASK SIGNAL
INTMSK<0>,!INTERRUPT MASK SIGNAL


MONMSK<0>,!MONITOR MASK SIGNAL
RHTIME<0>,!READ HARDENED TIMER SIGNAL
PROFLAG<2:0>,!PROGRAM FLAGS

INRPT<0>,!INTERRUPT SIGNAL FROM PIN TO ROM SEQUENCER

FLTINT<0>,!FAULT INTERRUPT SIGNAL
```

```
!FTSC.ILLEGALOPCODEDETECTOR        P219

ILLOPC<0>,!ILLEGAL OPCODE SIGNAL
FETCHMAX<15:0>,!MAXIMUM VALUE FOR FETCH OPCODES
STOREMAX<15:0>,!MAXIMUM VALUE FOR STORE OPCODES


!FTSC.OVERFLOW DIVIDE CHECK AND CARRY OUT STATUS FLIP FLOPS      P221
!(FTSC.OVFDIVCRYFF)

OVFF<0>,!OVERFLOW FLIP FLOP
CRYFF<0>,!CARRY OUT FLIP FLOP
DIVFF<0>,!DIVIDE STATUS FLIP FLOP
ARFLT<0>,!ARITHMETIC FAULT SIGNAL
FCHSTR<5:0>,!CPU FETCH/STORE CONTROL SIGNALS


!FTSC.GENERALPURPOSEFLIPFLOPS      P219

GPSF01<0>,!GENERAL PURPOSE FLIP FLOP 1
GPSF02<0>,!GENERAL PURPOSE FLIP FLOP 2
GPSF03<0>!GENERAL PURPOSE FLIP FLOP 3


**CPU.Buses**

!FTSC.CPUFETCHANDSTORE        P230

IMAB<15:0>,!CPU ADDRESS BUS
IMDB<31:0>!CPU DATA BUS


**Loop.Timer**

!FTSC.LOOPTIMERP219

SEQ11<0>,!LOOP BRANCH CONDITIONS
SEQ15<0>,
SEQ22<0>,
SEQ38<0>,
SEQ113<0>,

CPUCLK<0>,!CPU CLOCK PULSE
COUNTER<6:0>,!COUNTS UP TO 113 PULSES
ASUBFF<0>,!AUTOSUBTRACT FLIP FLOP
INTMUL<1:0>,!FOR AUTOMULTIPLY
FLOMUL<1:0>


**Processes**
```

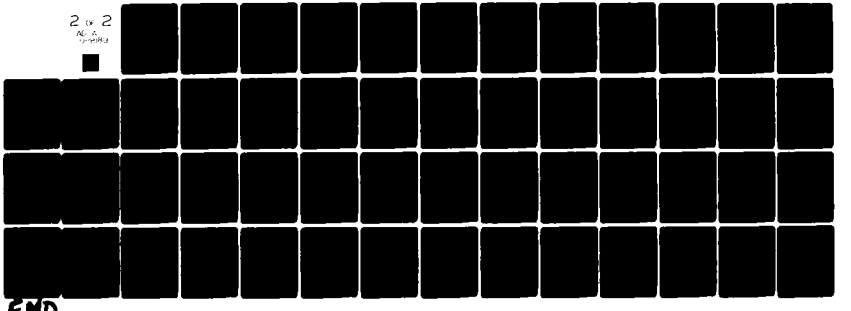
```
!FTSC.ROMSEQUENCER      P213.217

SEQUENCER:=

BEGIN
IF FLTINT=>
(RAD←0 NEXT!P115.218
IMOB←RECONF[GROM[9] NEXT
PERMSKREG<7>←1 NEXT
LEAVE SEQUENCER) NEXT
IF ILLOPC=>!P218
(RAD←2 NEXT LEAVE SEQUENCER)
NEXT
RAD<9:6>←RF02<9:6> NEXT
DECODE RF01 =>
BEGIN
0:=(RAD<5>←ROMA4;RAD<4>←ROMA5;RAD<3>←ROMA6;
    RAD<2>←ROMA7;RAD<1>←ROMA8;RAD<0>←ROMA9),
1:=(RAD<5>←INR<30>;RAD<4>←INR<29>;RAD<3>←INR<28>;
    RAD<2>←INR<27>;RAD<1>←INR<26>;RAD<0>←INR<25>),
2:=(RAD<5>←ROMA4;RAD<4>←EXG23;RAD<3>←OVF8;RAD<2>←SUM<0>;
    RAD<1>←ZDT8;RAD<0>←SUM<7>),
3:=(RAD<5>←ROMA4;RAD<4>←SUMM1;RAD<3>←ZDT24;RAD<2>←SUM<29>;
    RAD<1>←SUM<30>;RAD<0>←SUM<31>),
4:=(RAD<5>←ROMA4;RAD<4>←ROMA5;RAD<3>←INR<31>;RAD<2>←INR<18>;
    RAD<1>←INR<17>;RAD<0>←INR<16>),
5:=(RAD<5>←ROMA4;RAD<4>←ROMA5;RAD<3>←OVF8;RAD<2>←SUM<29>;
    RAD<1>←SUM<30>;RAD<0>←SUM<31>),
6:=(RAD<5>←ROMA4;RAD<4>←ROMA5;RAD<3>←ZDT24;RAD<2>←SUM<29>;
    RAD<1>←SUM<30>;RAD<0>←SUM<31>),
7:=(RAD<5>←ROMA4;RAD<4>←ROMA5;RAD<3>←ZDT32;RAD<2>←SUM<29>;
    RAD<1>←SUM<30>;RAD<0>←SUM<31>),
8:=(RAD<5>←ROMA4;RAD<4>←ROMA5;RAD<3>←OVF8;RAD<2>←SUM<27>;
    RAD<1>←SUM<29>;RAD<0>←SUM<28>),
9:=(RAD<5>←ROMA4;RAD<4>←ROMA5;RAD<3>←ROMA6;RAD<2>←GPSF03;
    RAD<1>←GPSF02;RAD<0>←GPSF01),
10:=(RAD<5>←ROMA4;RAD<4>←ROMA5;RAD<3>←ROMA6;RAD<2>←SUM<0>;
    RAD<1>←ZDT8;RAD<0>←SUM<7>),
11:=(RAD<5>←ROMA4;RAD<4>←ROMA5;RAD<3>←ROMA6;RAD<2>←ZDT32;
    RAD<1>←SUM<0>;RAD<0>←SUM<31>),
12:=(RAD<5>←ROMA4;RAD<4>←ROMA5;RAD<3>←ROMA6;RAD<2>←ROMA7;
    RAD<1>←SUMM1;RAD<0>←SUM<31>),
13:=(RAD<5>←ROMA4;RAD<4>←ROMA5;RAD<3>←ROMA6;RAD<2>←ROMA7;
    RAD<1>←ZDT24;RAD<0>←SUM<31>),
14:=(RAD<5>←ROMA4;RAD<4>←ROMA5;RAD<3>←ROMA6;RAD<2>←ROMA7;
    RAD<1>←ZDT32;RAD<0>←SUM<31>),
15:=(RAD<5>←ROMA4;RAD<4>←ROMA5;RAD<3>←ROMA6;RAD<2>←ROMA7;
    RAD<1>←SUM<0>;RAD<0>←SUM<1>),
16:=(RAD<5>←ROMA4;RAD<4>←ROMA5;RAD<3>←ROMA6;RAD<2>←ROMA7;
    RAD<1>←SUM<8>;RAD<0>←SUM<9>),
17:=(RAD<5>←ROMA4;RAD<4>←ROMA5;RAD<3>←ROMA6;RAD<2>←ROMA7;
    RAD<1>←SEQ113;RAD<0>←OVF8),
18:=(RAD<5>←ROMA4;RAD<4>←ROMA5;RAD<3>←ROMA6;RAD<2>←ROMA7;
```

```
      RAD<1>←CNTRL;RAD<0>←INRPT),
19:=(RAD<5>←ROMA4;RAD<4>←ROMA5;RAD<3>←ROMA6;RAD<2>←ROMA7;
      RAD<1>←ROMA8;RAD<0>←ZDT32),
20:=(RAD<5>←ROMA4;RAD<4>←ROMA5;RAD<3>←ROMA6;RAD<2>←ROMA7;
      RAD<1>←ROMA8;RAD<0>←SEQ11),
21:=(RAD<5>←ROMA4;RAD<4>←ROMA5;RAD<3>←ROMA6;RAD<2>←ROMA7;
       RAD<1>←ROMA8;RAD<0>←SEQ15),
22:=(RAD<5>←ROMA4;RAD<4>←ROMA5;RAD<3>←ROMA6;RAD<2>←ROMA7;
       RAD<1>←ROMA8;RAD<0>←SEQ22),
23:=(RAD<5>←ROMA4;RAD<4>←ROMA5;RAD<3>←ROMA6;RAD<2>←ROMA7;
       RAD<1>←ROMA8;RAD<0>←SEQ30),
24:=(RAD<5>←ROMA4;RAD<4>←ROMA5;RAD<3>←ROMA6;RAD<2>←ROMA7;
       RAD<1>←ROMA8;RAD<0>←SUM<7>),
25:=(RAD<5>←ROMA4;RAD<4>←ROMA5;RAD<3>←ROMA6;RAD<2>←ROMA7;
      RAD<1>←ROMA8;RAD<0>←OVFF),
26:=(RAD<5>←ROMA4;RAD<4>←ROMA5;RAD<3>←ROMA6;RAD<2>←ROMA7;
       RAD<1>←ROMA8;RAD<0>←CRYFF),

27:=(RAD<5>←ROMA4;RAD<4>←ROMA5;RAD<3>←ROMA6;RAD<2>←ROMA7;
      RAD<1>←ROMA8;RAD<0>←FDT32),
28:=(RAD<5>←ROMA4;RAD<4>←ROMA5;RAD<3>←ROMA6;RAD<2>←ROMA7;
      RAD<1>←ROMA8;RAD<0>←ZDT8),
29:=(RAD<5>←ROMA4;RAD<4>←ROMA5;RAD<3>←ROMA6;RAD<2>←ROMA7;
      RAD<1>←ROMA8;RAD<0>←MONMD),
30:=(RAD<5>←ROMA4;RAD<4>←ROMA5;RAD<3>←ROMA6;RAD<2>←ROMA7;
      RAD<1>←ROMA8;RAD<0>←SUM<31>),
31:=(RAD<5>←ROMA4;RAD<4>←ROMA5;RAD<3>←ROMA6;RAD<2>←ROMA7;
      RAD<1>←ROMA8;RAD<0>←AMODE)
END
NEXT
DECODE INR<18:16>=>
BEGIN
0:= AMODE←0,
OTHERWISE:=AMODE←1
END
END, !OF SEQUENCER

NEXTROMWORD:=
BEGIN
MICWORD1←CONTROM1[RAD] NEXT!NEXT ROMWORD
MICWORD2←CONTROM2[RAD] NEXT
MICWORD3←CONTROM3[RAD] NEXT
RF01<4:0>←MICWORD1<31:27>NEXT
RF02<9:0>←MICWORD1<26:17>NEXT
RF03<2:0>←MICWORD1<16:14>NEXT
RF04<2:0>←MICWORD1<13:11>NEXT
RF05<2:0>←MICWORD1<10:8>NEXT
RF06<2:0>←MICWORD1<7:5>NEXT
RF07<0>←MICWORD1<4>NEXT
RF08<0>←MICWORD1<3>NEXT
RF09<2:0>←MICWORD1<2:0>NEXT
RF10<2:0>←MICWORD2<31:29>NEXT
RF11<0>←MICWORD2<28> NEXT
```

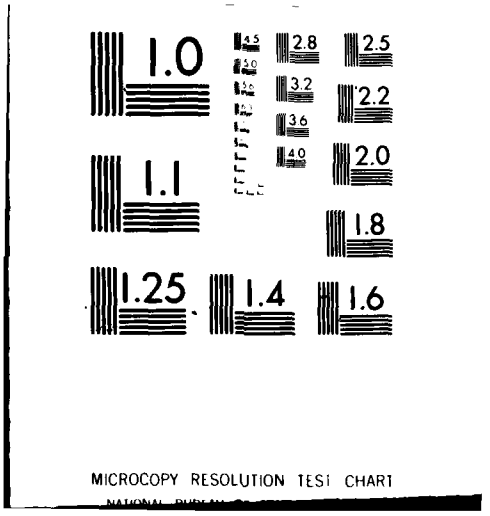
```
RF12<0>←MICWORD2<27> NEXT
RF13<0>←MICWORD2<26> NEXT
RF14<0>←MICWORD2<25> NEXT
RF15<2:0>←MICWORD2<24:22> NEXT
RF16<2:0>←MICWORD2<21:19> NEXT
RF17<3:0>←MICWORD2<18:15> NEXT
RF18<3:0>←MICWORD2<14:11> NEXT
RF19<0>←MICWORD2<10> NEXT
RF20<0>←MICWORD2<9> NEXT
RF21<0>←MICWORD2<8> NEXT
RF22<2:0>←MICWORD2<7:5> NEXT
RF23<0>←MICWORD2<4> NEXT
RF24<0>←MICWORD2<3> NEXT
RF25<0>←MICWORD2<2> NEXT

RF26<0>←MICWORD2<1> NEXT
RF27<0>←MICWORD2<0> NEXT
RF28<0>←MICWORD3<13>NEXT
RF29<0>←MICWORD3<12>NEXT
RF30<0>←MICWORD3<11>NEXT
RF31<4:0>←MICWORD3<10:6>NEXT
RF32<0>←MICWORD3<5>NEXT
RF33<0>←MICWORD3<4>NEXT
RF34<0>←MICWORD3<3>NEXT
RF35<0>←MICWORD3<2>NEXT
RF36<0>←MICWORD3<1>NEXT
RF37<0>←MICWORD3<0>
END,  !OF NEXTROMWORD


!FTSC.ROMFUNCTIONDECODER        P.220


DECODER:=
BEGIN
RFD00←0 NEXT
RFD01←0 NEXT
RFD02←0 NEXT
RFD03←0 NEXT
RFD04←0 NEXT
RFD05←0 NEXT
RFD06←0 NEXT
RFD07←0 NEXT
RFD08←0 NEXT
RFD09←0 NEXT
RFD10←0 NEXT
RFD11←0 NEXT
RFD12←0 NEXT
RFD13←0 NEXT
RFD14←0 NEXT
RFD15←0 NEXT
```

AD-A088 189 UNIVERSITY OF SOUTHERN CALIFORNIA MARINA DEL REY INFO--ETC F/G 9/2
MICROCODE VERIFICATION PROJECT.(U)
MAY 80 S D CROCKER, L MARCUS, D VAN-MIEROP F30602-78-C-0008
UNCLASSIFIED RADC-TR-80-42 NL

2 OF 2



END
8C


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
MICROCOPY RESOLUTION TEST CHART

```
RFD16~0 NEXT
RFD17~0 NEXT
RFD18~0 NEXT
RFD19~0 NEXT
RFD20~0 NEXT
RFD21~0 NEXT

RFD22~0 NEXT
RFD23~0 NEXT
RFD24~0 NEXT
RFD25~0 NEXT
RFD26~0 NEXT
RFD27~0 NEXT
RFD28~0 NEXT
RFD29~0 NEXT
RFD30~0 NEXT
RFD31~0 NEXT

DECODE RF31 =>
BEGIN
0:=RFD00~1,
1:=RFD01~1,
2:=RFD02~1,
3:=RFD03~1,
4:=RFD04~1,
5:=RFD05~1,
6:=RFD06~1,
7:=RFD07~1,
8:=RFD08~1,
9:=RFD09~1,
10:=RFD10~1,
11:=RFD11~1,
12:=RFD12~1,
13:=RFD13~1,
14:=RFD14~1,
15:=RFD15~1,
16:=RFD16~1,
17:=RFD17~1,
18:=RFD18~1,
19:=RFD19~1,
20:=RFD20~1,
21:=RFD21~1,
22:=RFD22~1,
23:=RFD23~1,
24:=RFD24~1,
25:=RFD25~1,
26:=RFD26~1,
27:=RFD27~1,
28:=RFD28~1,
29:=RFD29~1,
30:=RFD30~1,

31:=RFD31~1
```

```
END
END,!OF DECODER


!THE NEW VALUES OF MANOVF AND MANCOUT STILL HAVE TO BE CALCULATED
!ALONG WITH MANOUT BELOW.


MANOUT<24:0>:=
BEGIN
DECODE RF37=>
BEGIN
0:=(MANA25<24:0><=MANINA  NEXT MANB25<24:0><=MANINB NEXT
(DECODE MANSELECT =>
BEGIN
0:=MANCOUT@MANOUT←(MANA25<24:0> AND MANB25<24:0>),
1:=MANCOUT@MANOUT←(MANA25<24:0> EQV MANB25<24:0>),
2:=MANCOUT@MANOUT←(NOT MANA25<24:0>) + 0@MANCIN,
3:=MANCOUT@MANOUT←MANB25<24:0> + 0@MANCIN,
4:=MANCOUT@MANOUT←(NOT MANB25<24:0>) + 0@MANCIN,
5:=MANCOUT@MANOUT←MANA25<24:0> + 0@MANCIN,
6:=MANCOUT@MANOUT←MANA25<24:0> + MANB25<24:0>+0@MANCIN,
7:=MANCOUT@MANOUT←#177777777+(0@MANCIN),
8:=MANCOUT@MANOUT←MANB25<24:0>+NOT MANA25<24:0>+  0@MANCIN, !Sometimes it
!looks like MANA25<24:0> above should be just MANINA. Similarly in next line.
9:=MANCOUT@MANOUT←MANA25<24:0>+NOT MANB25<24:0>+0@MANCIN,
10:=MANCOUT@MANOUT←MANA25<24:0>+-(0@NOT MANCIN),
11:=MANCOUT@MANOUT←(NOT MANB25<24:0>)+ - (0@NOT MANCIN),
12:=MANCOUT@MANOUT←MANB25<24:0>+-(0@NOT MANCIN),
13:=MANCOUT@MANOUT←(NOT MANA25<24:0>)+ - (0@NOT MANCIN),
14:=MANCOUT@MANOUT←(MANA25<24:0> XOR MANB25<24:0>),
15:=MANCOUT@MANOUT←(MANA25<24:0> OR MANB25<24:0>)
END)),
1:=(MANA25<=MANINA NEXT
MANB25<=MANINB NEXT
IF ALUBLS=>MANB25←MANB25 SL0 1 NEXT
!FOR AUTOMULTIPLY SIGN-EXTEND  MANTISSA TWO BITS BEFORE SHIFTING
!BUT NOT EXPONENT?


(DECODE MANSELECT =>
BEGIN
0:=MANCOUT@MANOUT←(MANA25 AND MANB25),
1:=MANCOUT@MANOUT←(MANA25 EQV MANB25),
2:=MANCOUT@MANOUT←(NOT MANA25) + 0@MANCIN,
3:=MANCOUT@MANOUT←MANB25 + 0@MANCIN,
4:=MANCOUT@MANOUT←(NOT MANB25) + 0@MANCIN,
5:=MANCOUT@MANOUT←MANA25 + 0@MANCIN,
6:=MANCOUT@MANOUT←MANA25 + MANB25+0@MANCIN,
7:=MANCOUT@MANOUT←#177777777+(0@MANCIN),
8:=MANCOUT@MANOUT←MANB25+NOT MANA25+  0@MANCIN, !Sometimes it
!looks like MANA25 above should be just MANINA. Similarly in next line.
9:=MANCOUT@MANOUT←MANA25+NOT MANB25+0@MANCIN,
```

```
10:=MANCOUT@MANOUT←MANA25+-(0@NOT MANCIN),
11:=MANCOUT@MANOUT←(NOT MANB25)+ - (0@NOT MANCIN),
12:=MANCOUT@MANOUT←MANB25+-(0@NOT MANCIN),
13:=MANCOUT@MANOUT←(NOT MANA25)+ - (0@NOT MANCIN),
14:=MANCOUT@MANOUT←(MANA25 XOR MANB25),
15:=MANCOUT@MANOUT←(MANA25 OR MANB25)
END))
END
END,!OF MANOUT



!THE NEW VALUES OF EXPOVF AND EXPCOUT MUST STILL BE CALCULATED ALONG
!WITH EXPOUT BELOW. SEE LINES 8-13 BELOW.

EXPOUT<8:0>:=
BEGIN
EXPA9<=EXPINA NEXT
EXPB9<=EXPINB NEXT
IF ALUBLS=>EXPB9←EXPB9 SL0 1 NEXT
(DECODE EXPSELECT =>
BEGIN
0:=EXPCOUT@EXPOUT←EXPA9 AND EXPB9,
1:=EXPCOUT@EXPOUT←EXPA9 EQV EXPB9,
2:=DECODE EXPCIN=>(EXPCOUT@EXPOUT←NOT EXPA9,
                   EXPCOUT@EXPOUT←NOT EXPA9 + 0@RF19 +0@1),
3:=DECODE EXPCIN=>(EXPCOUT@EXPOUT←EXPB9,
                    EXPCOUT@EXPOUT←EXPB9 +0@RF19 +0@1),
4:=DECODE EXPCIN=>(EXPCOUT@EXPOUT←NOT EXPB9,
                   EXPCOUT@EXPOUT←NOT EXPB9 +0@RF19 +0@1),
5:=DECODE EXPCIN=>(EXPCOUT@EXPOUT←EXPA9,
                   EXPCOUT@EXPOUT←EXPA9 +0@RF19 +0@1),

6:=EXPCOUT@EXPOUT←EXPA9 + EXPB9 + 0@EXPCIN,
7:=EXPCOUT@EXPOUT←#777+(0@EXPCIN),
8:=EXPCOUT@EXPOUT←EXPB9+NOT EXPA9 + 0@EXPCIN,
9:=EXPCOUT@EXPOUT←EXPA9+NOT EXPB9 + 0@EXPCIN,
10:=DECODE EXPCIN=>(EXPCOUT@EXPOUT←EXPA9+-(0@RF19+1),
                    EXPCOUT@EXPOUT←EXPA9),
11:=DECODE EXPCIN=>(EXPCOUT@EXPOUT←NOT EXPB9+-(0@RF19+1),
                    EXPCOUT@EXPOUT←NOT EXPB9),
12:=DECODE EXPCIN=>(EXPCOUT@EXPOUT←EXPB9+-(0@RF19+1),
                    EXPCOUT@EXPOUT←EXPB9),
13:=DECODE EXPCIN=>(EXPCOUT@EXPOUT←NOT EXPA9+-(0@RF19+1),
                    EXPCOUT@EXPOUT←NOT EXPA9),
14:=EXPCOUT@EXPOUT←EXPA9 XOR EXPB9,
15:=EXPCOUT@EXPOUT←EXPA9 OR EXPB9
END)
NEXT
OVF8←EXPOVF←EXPOUT<8> XOR EXPOUT<7>
END,!OF EXPOUT
```

```
!FTSC.FSFG(FLAG AND SPECIAL FUNCTION GENERATOR)
!P.226,94


FSFG:=!FLAG AND SPECIAL FUNCTION GENERATOR
   BEGIN
MONMSK←RHTIME←PROFLAG←0 NEXT
EXTADD←ROMADD←HSW1EN←HSW2EN←PERMSK←INTMSK←0 NEXT
DECODE IMAB =>
BEGIN
"F800:= HSW1EN←1,
"F801:= HSW2EN←1,
"F802:= MONMSK←1,
"F803:= PERMSK←1,
"F804:= INTMSK←1,
"F805:= RHTIME←1,
!IF IMAB GTR "F805 AND IMAB LSS "F809 => ERROR?
"F809:=PROFLAG←IMAB<2:0>,
"F80A:=PROFLAG←IMAB<2:0>,
"F80B:=PROFLAG←IMAB<2:0>,
"F80C:=PROFLAG←IMAB<2:0>,
"F80D:=PROFLAG←IMAB<2:0>,
"F80E:=PROFLAG←IMAB<2:0>,
"F80F:=PROFLAG←IMAB<2:0>,
OTHERWISE:=!SEE P 69
((IF ((IMAB GEQ "F000) AND (IMAB LEQ "F7FF))=>
ROMADD←1) ;
 (IF ((IMAB LSS "F000) OR (IMAB GTR "F7FF))=>
EXTADD←1))
END
   END,!OF FSFG


!FTSC.ILLEGALOPCODEDETECTOR        P219


DETECTOR:=
BEGIN

IF RFD12=>(ILLOPC←0 NEXT LEAVE DETECTOR) NEXT
IF (RFD01 OR RFD11)=>
(ILLOPC←IMDB<20> NEXT LEAVE DETECTOR) NEXT
!OR IMDB<21>? ON P64 IT SAYS BIT11=IMDB<20>
!BUT ON 205 IT SHOWS BIT10=IMDB <21> AS INPUT.

DECODE INR<31>=>
BEGIN
0:= IF 0=INR<30:25> GTR FETCHMAX =>ILLOPC←1,
1:= IF 0=INR<30:25> GTR STOREMAX =>ILLOPC←1
END
END,!OF DETECTOR
```

```
LOOP:=
BEGIN
PENDING←PENDING OR INTREQFF NEXT
(IF REQPRIORITY LEQ PRIORITYLEVEL =>LEAVE LOOP) NEXT
PRIORITYLEVEL ← REQPRIORITY NEXT
(IF PRIORITYLEVEL LEQ INPROCFF => LEAVE LOOP) NEXT
(IF ENADISFF => LEAVE LOOP) NEXT
INRPT←1 NEXT
IF RF27 =>INTREQREG<PRIORITYLEVEL> ← 0 NEXT
INPROCFF<PRIORITYLEVEL> ← 1
END,!OF LOOP


PRIORITY:=
BEGIN
REPEAT
BEGIN
REQPRIORITY ← REQPRIORITY + 1 NEXT
(IF (INTREQFF SR0 REQPRIORITY) EQL 1 =>
LEAVE PRIORITY) NEXT
(IF (REQPRIORITY EQL 8) => (REQPRIORITY ←0 NEXT
LEAVE PRIORITY))
END
END,     !REQPRIORITY=the level of highest interrupt requested.


!FTSC.PIN    (PRIORITY INTERRUPT NETWORK) P229 FF


PIN:=
BEGIN
IF PERMSK=>PERMSKREG←IMDB NEXT
IF FLTINT=>PERMSKREG<7>←1 NEXT
IF ARFLT=>INTREQREG<7>←1 NEXT
!ADD THE OTHER PRIORITIES HERE.
IF PERMSKREG<7> =>INTREQFF@INPROCFF←0 NEXT
IF INTMSK =~ INTMSKREG←IMDB NEXT
IF NOT PERMSKREG<7> =>INTREQFF ← (INTREQREG AND NOT INTMSKREG) NEXT
!    IF FLTINT =>... LEAVE PIN NEXT

!    IF ILLOPC =>... LEAVE PIN NEXT

REQPRIORITY<=-1 NEXT
PRIORITY() NEXT
LOOP() NEXT
IF (RFD10 OR RF27 OR PERMSKREG<7>) => INPROCFF←0 NEXT!ALL OF THEM.
!THERE STILL MAY BE SOME IN PENDING

IF (RFD01 OR RFD10)=>ENADISFF←IMDB<23> NEXT
IF RFD13 =>ENADISFF←1 NEXT
IF RFD12 =>ENADISFF←0
END,!OF PIN
```

```
!FTSC.LOOPTIMERP219


TIMER:=
BEGIN
COUNTER←COUNTER +1 NEXT
IF RFD14 =>(COUNTER←SEQ11←SEQ15←SEQ22←SEQ30←SEQ113 ←0) NEXT


DECODE COUNTER =>
BEGIN
11:=SEQ11←1,
15:=SEQ15←1,
22:=SEQ22←1,
30:=SEQ30←1,
       113:=SEQ113←1,
       114:=COUNTER←0   !Maybe not; or maybe need two counters:
!one for setting SEQ and one for counting microsteps.
END
END!OF TIMER


**Microinstruction.Cycle**

CYCLE(MAIN):=
BEGIN
DELAY(1) NEXT
RAD←1 NEXT
FETCHMAX←"35 NEXT    !THESE ARE THE MAXIMUM OP-CODES FOR

     !INR<31>=0,1 RESPECTIVELY
STOREMAX←"68 NEXT
COUNTER←0 NEXT
REPEAT
BEGIN
NEXTROMWORD() NEXT
SEQUENCER() NEXT
DECODER() NEXT
IF RFD21=>(GPSF01←GPSF02←GPSF03←ASUBFF←0) NEXT
!AUTO MULTIPLY FN. P224

DECODE RF37=>(AUTOMULFN←0,
(DECODE RF23=>
BEGIN
0:=MULBITS←INTMUL,!INTEGER FORMAT
1:=MULBITS←FLOMUL!FLOATING POINT FORMAT
END
NEXT
DECODE CRYSTS@MULBITS =>
BEGIN
0:=AUTOMULFN←2,
```

```
1:=AUTOMULFN←0,
2:=AUTOMULFN←#15,
3:=AUTOMULFN←#11,
4:=AUTOMULFN←0,
5:=AUTOMULFN←4,
6:=AUTOMULFN←#11,
7:=AUTOMULFN←#13
END) )
NEXT


MANWX←MANWR[RF03] NEXT
MANWRYB←MANWR[RF05] NEXT
EXPWX←EXPWR[RF04] NEXT
EXPWRYB←EXPWR[RF06] NEXT
!FTSC.GENERALPURPOSEREGISTERS    P209



!FTSC.ALUINPUTSELECTOR                P.208

!MANTISSA

DECODE RF20 =>
BEGIN
0:=MANINA←MANGPR[RA],

1:=MANINA←MANGPR[RB]
END
NEXT
DECODE ALUBZ =>
BEGIN
0:=DECODE RF21 OR AMODE=>(MANINB←MANGPR[RA],MANINB←MANWX),
1:=MANINB←0
END
NEXT

!EXPONENT

DECODE RF20 =>
BEGIN
0:=EXPINA←EXPGPR[RA],
1:=EXPINA←EXPGPR[RB]
END
NEXT
DECODE ALUBZ =>
BEGIN
0:=DECODE RF21 OR AMODE=>(EXPINB←EXPGPR[RA],EXPINB←EXPWX),
1:=EXPINB←0
END
NEXT
```

```
!FTSC.FUNCTIONINVERSION          P.207

INVERTOR←(RF36 AND INVERTOR) OR (RF37 AND INVERTOR) NEXT

DECODE RF23=>
BEGIN
0:=(INVEFN←INVERTOR; INVMFN←INVERTOR),
1:=INVMFN←INVERTOR
END
NEXT
DECODE INVEFN =>
BEGIN
0:=(EXPCIN←RF24;EXPSELECT←RF18),
1:=(EXPCIN←NOT RF24;EXPSELECT←NOT RF18)
END
NEXT

!FTSC.ALUFUNCTIONSELECTOR          P.206

!MANTISSA  ALU OUTPUT FUNCTION  (IN2=0)



!EXPONENT ALU OUTPUT FUNCTION



EXPOUT() NEXT

IF ASUBFF =>(INVMFN←RF33) NEXT!INVERT CARRY IN BITS TO MANTISSA.SEE P 222
DECODE INVMFN =>
BEGIN
0:=(DECODE RF23 =>
 BEGIN
 0:=MANCIN← EXPCOUT,
 1:=MANCIN←RF25
 END;
   MANSELECT←RF17),
1:=(DECODE RF23 =>
BEGIN
0:=MANCIN← EXPCOUT, !Note this deviation(?) from the documentation.
1:=MANCIN←NOT RF25
END;
   MANSELECT←NOT RF17)
END
NEXT

MANOUT() NEXT
!FTSC.ALUOUTPUTS          P204
```

```
IMDB←SUM←MANOUT<23:0>@EXPOUT<7:0> NEXT   !Here I assume that
!SUMLSBEN,SUMMSBEN are always on so that
!any output of the ALU goes to IMDB and SUM.
SUMM1←MANOUT<24> NEXT



SUMM2←MANCOUT NEXT
DECODE SUM<31:8>=>
BEGIN
0:=ZDT24←1,
OTHERWISE:=ZDT24←0
END
NEXT

DECODE SUM<7:0>=>
BEGIN
0:=ZDT8←1,
OTHERWISE:=ZDT8←0
END
NEXT

DECODE SUM=>
BEGIN
0:=(ZDT32←1;FDT32←0),
#37777777777:=(FDT32←1;ZDT32←0),
OTHERWISE:=(FDT32←ZDT32←0)
END
NEXT
EXG23←0 NEXT
IF (SUM<7:0> GTR 23) OR (SUM<7:0> LSS -23) => EXG23←1 NEXT
!FTSC.ENDCONDITIONSGENERATOR      P223


!NOTE THE QUESTION MARKS BELOW!

DECODE RF22 =>
BEGIN
0:=ENDCONDS←SUM<0>@SUM<1>@WRYB<31>@WRYB<30>@SUM<31>@SUM<30>
        @WRYB<31>@WRYB<31>@(RF25 XOR INVMFN)@'0@WRYB<31>@'00,
1:=ENDCONDS←SUMM1@SUMM2@WRYB<31>@WRYB<30>@'00@SUM<8>
        @SUM<9>@(RF24 XOR INVEFN)@'0@SUM<8>@'00,
!THERE IS STILL SOME DOUBT IF THE ABOVE LINE IS CORRECT.  SEE P223
2:=ENDCONDS←'1@SUM<31>@SUM<7>@SUM<6>@SUM<31>@SUM<30>
          @SUM<8>@SUM<9>@(RF25 XOR INVMFN)@'0@SUM<8>@'01,
3:=ENDCONDS←'00@SUM<7>@SUM<6>@'00@SUM<8>@SUM<9>
        @(RF25 XOR INVMFN)@'0@SUM<8>@SUM<31>@SUM<30>,
4:=ENDCONDS←WRYB<0>@WRYB<1>@SUM<7>@SUM<6>@WRYB<31>@WRYB<30>
     @SUM<0>@SUM<1>@WRYB<7>@WRYB<6>@SUM<0>@(RF24 XOR INVEFN)@'1,
5:=ENDCONDS←SUMM1@SUMM2@SUM<7>@SUM<6>@WRYB<31>@WRYB<30>
        @SUM<0>@SUM<1>@WRYB<7>@WRYB<6>@SUM<0>@'00,
```

```
6:=ENDCONDS←SUM<31>⊕SUM<31>⊕SUM<7>⊕SUM<6>⊕WRYB<31>⊕WRYB<30>
        ⊕SUM<0>⊕SUM<1>⊕WRYB<7>⊕WRYB<6>⊕SUM<0>⊕'11,
7:=ENDCONDS←'00⊕SUM<7>⊕SUM<6>⊕WRYB<31>⊕WRYB<30>⊕SUM<0>
        ⊕SUM<1>⊕WRYB<7>⊕WRYB<6>⊕SUM<0>⊕SUM<31>⊕SUM<30>
END
NEXT



!AUTODIVIDE FUNCTION     P.225

IF RF36=>(INVERTOR←SUM<31>) NEXT!PREVIOUS SUM


!AUTOSUBTRACT FUNCTION

IF RFD22=>(ASUBFF←1) NEXT


IF RFD07=>GPSF01←1 NEXT
IF RFD08=>GPSF02←1 NEXT
IF RFD09=>GPSF03←1 NEXT


!FTSC.CPUFETCHANDSTORE        P230


IF RF26=>!MEMORY REQUEST("SPEED UP")
(DECODE RF32=>
BEGIN!ADDRESS
0:=!MAB←WRYB<15:8>⊕EXPWRYB,
1:=!MAB←(WRYB<15:8>⊕EXPWRYB) + #10000     !ADD 4096
END) NEXT

IF LOBMRAR => MRAR←IMAB NEXT
!THE FLAG AND SPECIAL FUNCTION GENERATOR COMES HERE (FTSC.FSFG)
!(STILL INSIDE IF RF26=>?)
!SINCE IT COMPUTES THE VALUE OF EXTADD WHICH IS NEEDED BELOW.
FSFG() NEXT


DETECTOR() NEXT



!  CRYFF←0 NEXT!OR IS RESTORING ENOUGH? SEE BELOW
!  DIVFF←0 NEXT!DITTO
IF RFD03=>OVFF←1 NEXT
IF RFD04=>OVFF←0 NEXT
IF RFD05=>CRYFF←MANCOUT NEXT
```

```
IF RFD06 =>DIVFF←1 NEXT
IF RFD10=>(OVFF←IMDB<21> NEXT!RESTORING
   CRYFF←IMDB<19> NEXT
   DIVFF←IMDB<22>) NEXT
IF RF28=>(IMDB←ENADISFF⊕DIVFF⊕OVFF⊕ILLOPC⊕CRYFF
              ⊕PRIORITYLEVEL<2:0>⊕MANPC<7:0>⊕EXPPC) NEXT
ARFLT←DIVFF OR OVFF NEXT
FCHSTR←RF26⊕RF27⊕RF28⊕RF30⊕RFD10⊕EXTADD NEXT
        !NOTE ORDER IS DIFFERENTTHAN ON 230
           !Is EXTADD set in FSFG before type of address is known?

DECODE FCHSTR=>
BEGIN
#45:= IMDB←MEM[IMAB],!NORMAL FETCH (INR<31>=0)
#41:= MEM[IMAB]←IMDB,!NORMAL STORE(INR<31>=1)
#44:=  (IF HSW1EN =>IMDB←HSW1⊕MRAR NEXT !CPU FETCH P226,P88
IF HSW2EN =>IMDB←HSW2 NEXT!P90
IF MONMSK =>IMDB←MONMSKREG NEXT!P87
IF PERMSK =>IMDB←PERMSKREG NEXT
IF INTMSK =>IMDB←INTMSKREG),!P68


!ET CETERA,

#40:=!CPU STORE
(IF HSW1EN =>HSW1←IMDB<31:16> NEXT
IF HSW2EN => HSW2←IMDB NEXT
IF MONMSK =>MONMSKREG←IMDB NEXT
IF PERMSK =>PERMSKREG←IMDB NEXT
IF INTMSK =>INTMSKREG←IMDB),

!ET CETERA,
#24:#25:=(IMAB←"F000 + PRIORITYLEVEL NEXT
            !"VECTOR JUMP"= INTVEC ON 216.
IMDB←MEM[IMAB]),
!PIN SENDS OUT ADDRESS OF INTERRUPT SERVICE ROUTINE; SEE 73.
#10:#11:=(IMAB←PRIORITYLEVELNEXT!"JSB1"= INTRET ON 216.
MEM[IMAB]←IMDB),
!0:1:=!JSB2
!SPC1 IS SAME DECODE VALUE AS JSB1
!SPC2 IS SAME DECODE VALUE AS NORMAL STORE
#47:=IMDB←MEM[IMAB]!RFI
!RET IS SAME DECODE VALUE AS RFI

END
NEXT



!FTSC.GENERALPURPOSEFLIPFLOPS      P219
```

```
PIN() NEXT
DECODE RF00=>
BEGIN              !P.210
0:=MANGPRIN=SUM<31:8>,
1:=MANGPRIN=SUM2RS@SUMMRS@SUM<31:10>,
2:=MANGPRIN=0,
3:=MANGPRIN=IMDB<31:8>,
4:=MANGPRIN=SUM<30:8>@SUMMLS,
5:=MANGPRIN=SUMMRS@SUM<31:9>,
6:=MANGPRIN=SUM<29:8>@SUMMLS@SUM2LS,
7:=MANGPRIN<=IMDB<15:8>
END
NEXT
DECODE RF10=>
BEGIN
0:=EXPGPRIN=SUM<7:0>,
1:=EXPGPRIN=SUM<9:2>,
2:=EXPGPRIN=0,
3:=EXPGPRIN=IMDB<7:0>,
4:=EXPGPRIN=SUM<6:0>@SUMELS,
5:=EXPGPRIN=SUM<3:1>,
6:=EXPGPRIN=SUM<5:0>@SUMELS@SUME2LS,
7:=EXPGPRIN=IMDB<7:0>
END

NEXT

IF RF11=> (DECODE RF20=>
BEGIN
0:=MANGPR[RA]=MANGPRIN,
1:=MANGPR[RB]=MANGPRIN
END) NEXT
IF RF12=> (DECODE RF20=>
BEGIN
0:=EXPGPR[RA]=EXPGPRIN,
1:=EXPGPR[RB]=EXPGPRIN
END) NEXT
    !FTSC.WORKINGREGISTERS          P.209


DECODE RF15=>
BEGIN
0:=MANWRIN=WRYB<31:8>,
1:=MANWRIN=WRYM2RS=WRYMRS@WRYB<31:10>,
2:=MANWRIN=SUM<31:8>,
3:=MANWRIN=IMDB<31:8>,
4:=MANWRIN=WRYB<30:8>@WRYMLS,
5:=MANWRIN=WRYMRS@WRYB<31:9>,
6:=MANWRIN=WRYB<29:8>@WRYMLS@WRYM2LS,
7:=MANWRIN<=IMDB<15:8>
END
NEXT
```

```
DECODE RF16=>
BEGIN
0:=EXPWRIN←WRYB<7:0>,
1:=EXPWRIN←WRYB<9:2>,
2:=EXPWRIN←SUM<7:0>,
3:=EXPWRIN←IMDB<7:0>,
4:=EXPWRIN←WRYB<6:0>@WRYELS,
5:=EXPWRIN←WRYB<8:1>,
6:=EXPWRIN←WRYB<5:0>@WRYELS@WRYE2LS,
7:=EXPWRIN←IMDB<7:0>
END

NEXT
IF RF13=>
(DECODE RF07=>
BEGIN
0:=MANWR[RF05]←MANWRIN,
1:=MANWR[RF03]←MANWRIN
END) NEXT
IF RF14=>
(DECODE RF08=>
BEGIN
0:=EXPWR[RF06]←EXPWRIN,
1:=EXPWR[RF04]←EXPWRIN
END) NEXT


DECODE RF37=>(INTMUL←FLOMUL←0,
(INTMUL←WRYB<3:2> NEXT
FLOMUL←WRYB<11:10>)) NEXT


!FTSC.INSTRUCTIONREGISTERP209,213


IF RFD20 => INR←IMDB NEXT
IF RFD15 =>(RA←RA + 1 NEXT
    RB←RB + 1) NEXT!WHAT HAPPENS IF RA OR RB GETS TOO LARGE?
IF RF35 => RB←RB + 1 NEXT
TIMER()


END!OF REPEAT IN CYCLE
END!OF CYCLE
)!END OF MICROFTSC
'
```

Appendix C
FTSC TARGET

```
FTSC TARGET


MACROFTSC:=(


**Memory**

MEM[0:32K]<31:0>

**Registers**

COUNTER<31:0>,          !Loop counter

!WATCH OUT: THE COUNTER HERE IS NOT THE SAME AS IN FTSC.MIC!
GPXR[0:7]<31:0>,                !8 general purpose registers
W0<31:0>,                       !Working register 0
W1<31:0>,                       !Working register 1
W2<31:0>,                       !Working register 2
W3<31:0>,                       !Working registr 3
EX<31:0>,                       !Extension register
MD<31:0>,                       !Memory data
MA<31:0>,                       !Memory address
PC<31:0>,                       !Program counter
EXPOUT<8:0>,                    !9-bit output of exponent ALU
SUM<31:0>,                      !32-bit output of ALU
ALUA<33:0>,
ALUB<33:0>,
EXPA9<8:0>,
EXPB9<8:0>,
INTPRIOR<31:0>,         !highest pending interrupt level

INR<31:0>,
         AMODE<2:0>:=INR<18:16>,
         RA<2:0>:= INR<21:19>,
         RB<2:0>:=INR<24:22>,
         OPCODE<6:0>:=INR<31:25>,
        MACRO GPXRA:=GPXR[RA]!,
        MACRO GPXRB:=GPXR[RB]!


**Signals**

OVFF<0>,
```

```
CRYSTS<0>,
SUMM2<0>,
SUMM1<0>,
OVF8<0>,
DIVFF<0>,
CRYFF<0>,
INVERTOR<0>,
EXG23<0>,
INRPT<0>,
MON<0>,
ASUBE<0>,
EXMODE<0>,                          !Executive mode
ILLOPC<0>,
DISINT<0>,                          !disable interrupt
        MACRO STATUS:=!EXMODE&DISINT&DIVFF&OVFF&ILLOPC&CRYFF&INTPRIOR<2:0>!

**Addressing.Fetching**

INSTRUCTION:=
        BEGIN
        INR←MEM[PC] NEXT
        MA<=INR<15:0> NEXT
        PC←PC+1
        END,

ADDRESS:=
        BEGIN
        DECODE AMODE=>
                BEGIN
                0:2:=NO.OP(),           !Reg-reg, immediate, direct
                  3:=MA←MEM[MA],                !indirect
                  4:=(MA←MA+GPXRA NEXT          !indexed, post-increment
                     GPXRA←GPXRA+1),
                  5:=(GPXRA←GPXRA-1 NEXT        !indexed, pre-decrement
                      MA←MA+GPXRA),
                  6:=MA←MA+GPXRA,               !indexed
                  7:=MA←MEM[MA+GPXRA]           !indexed, indirect
                END
        END,


OPERAND:=
        BEGIN
        IF NOT INR<31>=>
                (DECODE AMODE=>
                        BEGIN
                        0:=MD←GPXRA,  !This is slightly different from
! the real machine: there AMODE is checked in each function and sometimes
! even if AMODE = 0, GPXRA does not have to go through MD.

!SO THERE'S NO NEED FOR ALL THE "DECODE AMODE"'S IN THE BODY OF THE PROGRAM!
!BUT MAYBE IT'S BETTER TO LEAVE THEM IN, AND ELIMINATE THE (THEN) EXTRANEOUS
!DECODE AMODE IN OPERAND, IN ORDER TO MAKE THE AUTOMATIC PROVING EASIER.
```

```
!OR INDEED IN ORDER TO MAKE IT POSSIBLE:  IF THE MACRODESCRIPTION SAYS
!MD←GPXRA BUT IN FACT THAT DOES NOT HAPPEN , THEN IT CANNOT BE PROVED.
!WE COULD INTRODUCE ANOTHER VARIABLE "ARG" TO TAKE THE PLACE OF
!"GPXRA PHI MD".
                        1:=MD←MA,
                OTHERWISE:= MD←MEM[MA]
                        END)
        END

☆☆Processes☆☆


! THE COMPLETE INSTRUCTION CYCLE IS CODED UNTIL CONTROL RETURNS TO INR FETCH
!  OR "ALPHA", DEPENDING ON THE INSTRUCTION.  THIS DIFFERENCE WILL HAVE
 ! TO BE COMPENSATED FOR LATER.

LDR:=
        BEGIN
        GPXRB←MD
        END.
LDE:=
        BEGIN
        EX←MD
        END.

!LW0-LW3 ARE NOT CODED, BUT IF NEEDED CAN BE CODED LIKE LDR AND LDE.

LOOP1:=
        BEGIN
        MA←MA+1 NEXT
        DECODE AMODE=>
                BEGIN
                0:=GPXRB←GPXRA,
    OTHERWISE:=GPXRB←MEM[MA]
                END
        END .


LDR2:=
        BEGIN
        LDR() NEXT
        RA←RA+1 NEXT
        RB←RB+1 NEXT
        LOOP1()
        END.

LOOP2:=
        BEGIN
        LOOP1() NEXT
        RA←RA+1 NEXT
        RB←RB+1
        END.
```

```
LDR3:=
        BEGIN
        LDR() NEXT
        RA←RA+1 NEXT
        RB←RB+1 NEXT
        LOOP2() NEXT
        LOOP1()
        END,

LDR7:=
        BEGIN
        LDR() NEXT
        LOOP2() NEXT
        LOOP2() NEXT
        LOOP2() NEXT
        LOOP2() NEXT
        LOOP2() NEXT
        LOOP1()
        END,

LDN:=
        BEGIN
        ALUB<32:0><-MD NEXT
        SUMM1⊕SUM←-ALUB NEXT
        GPXRB←SUM NEXT
        IF SUMM1 XOR SUM<31>=>OVFF←1      !OVERFLOW DETECTION
        END,


!  From here to the end of NORMAL has been checked with DIVF, Mar.8,78.
! as DIVFML.

NMLOOP:=
        BEGIN
        REPEAT
        BEGIN
        IF OVF8=>(OVFF←1 NEXT LEAVE NMLOOP) NEXT
        DECODE GPXRB<29:27>=>
                BEGIN
                [0,7]:=(EXPOUT←(GPXRB<7>⊕GPXRB<7:0>)+-2 NEXT
                                OVF8←EXPOUT<8> XOR EXPOUT<7> NEXT
                                GPXRB←GPXRB<29:8>⊕'00⊕EXPOUT<7:0>),
                2:5:=LEAVE NMLOOP,
                [1,6]:=(EXPOUT←(GPXRB<7>⊕GPXRB<7:0>)+-1 NEXT
                                OVF8←EXPOUT<8> XOR EXPOUT<7> NEXT
                                GPXRB←GPXRB<30:8>⊕'0⊕EXPOUT<7:0> NEXT
                                IF OVF8=>OVFF←1 NEXT
                                LEAVE NMLOOP)
                END
        END
```

```
        END,
NORMAL:=        !CALLED IN ADDF, SUBF,DIVF, LDAF,LDNF.
!Make sure that a test for OVF8 is made at the end of above instructions
!before entry into NORMAL.
        BEGIN
IF SUMM1⊗SUM<31:8> EQL 0=>(GPXRB←"80 NEXT
                                        LEAVE NORMAL) NEXT
        DECODE OVF8⊗SUMM1⊗SUM<31:29>=>
                BEGIN
               [#10:#13]:=(EXPOUT←(GPXRB<7>⊗GPXRB<7:0>)+1 NEXT
                                GPXRB←'1⊗GPXRB<31:9>⊗EXPOUT<7:0> NEXT
                                OVF8←EXPOUT<8> XOR EXPOUT<7> NEXT
                                IF  OVF8=>OVFF←1),
! THAT'S RIGHT: IF BOTH THE PREVIOUS EXPONENT AND THE PRESENT ONE
! OVERFLOW THEN THERE IS NO GENERAL OVERFLOW.
                [#30:#33]:=(EXPOUT←(GPXRB<7>⊗GPXRB<7:0>)+1 NEXT
                                GPXRB←'1⊗GPXRB<31:9>⊗EXPOUT<7:0> NEXT
                                OVF8←EXPOUT<8> XOR EXPOUT<7> NEXT
                                IF NOT OVF8=>OVFF←1),
                [4:7]:=(EXPOUT←(GPXRB<7>⊗GPXRB<7:0>)+1 NEXT
                                GPXRB←'0⊗GPXRB<31:9>⊗EXPOUT<7:0> NEXT
                                OVF8←EXPOUT<8> XOR EXPOUT<7> NEXT
                                IF  OVF8=>OVFF←1),
                [#24:#27]:=(EXPOUT←(GPXRB<7>⊗GPXRB<7:0>)+1 NEXT
                                GPXRB←'0⊗GPXRB<31:9>⊗EXPOUT<7:0> NEXT
                                OVF8←EXPOUT<8> XOR EXPOUT<7> NEXT
                                IF NOT OVF8=>OVFF←1),
                [1,#16]:=(EXPOUT←(GPXRB<7>⊗GPXRB<7:0>)+-1 NEXT
                                OVF8←EXPOUT<8> XOR EXPOUT<7> NEXT
                                GPXRB←GPXRB<30:7>⊗W1<31>⊗EXPOUT<7:0> NEXT
!  WHAT DOES W1 CONTAIN IN ALL THE CASES WHERE NORMAL IS CALLED?
! IT LOOKS LIKE IN ALL THE ABOVE CASES W1=0. NO; THERE'S AT LEAST
! ONE CASE FROM ADDF WHERE W1 IS NOT ZERO.
                                IF  OVF8=>OVFF←1),
                [#21,#36]:=(EXPOUT←(GPXRB<7>⊗GPXRB<7:0>)+-1 NEXT
                                OVF8←EXPOUT<8> XOR EXPOUT<7> NEXT
                                GPXRB←GPXRB<30:7>⊗W1<31>⊗EXPOUT<7:0> NEXT
                                IF NOT OVF8=>OVFF←1),
        [#20,#37,0,#17]:=(EXPOUT←(GPXRB<7>⊗GPXRB<7:0>)+-2 NEXT
                                OVF8←EXPOUT<8> XOR EXPOUT<7> NEXT
                                GPXRB←GPXRB<29:7>⊗W1<31:30>⊗EXPOUT<7:0> NEXT
                                NMLOOP()),

            [#22,#23,#34,#35]:= IF OVF8=>OVFF←1,
                [2:3,#14:#15]:=IF OVF8=>OVFF←1
                END
        END,
! From NMLOOP to here has been checked with DIVF

LDNF:=
        BEGIN
        SUMM1⊗SUM<31:8>←-(MD<31>⊗MD<31:8>) NEXT
        GPXRB←SUM<31:8>⊗MD<7:0> NEXT
```

```
        W1←0 NEXT
        NORMAL()
        END


LDA:=
        BEGIN
        LDR() NEXT
        IF GPXRB<31>=>
                (SUMM1@SUM←-(GPXRB<31>@GPXRB) NEXT
                GPXRB←SUM NEXT
                IF SUMM1 XOR SUM<31>=>OVFF←1)
        END,

LDAF:=
        BEGIN
        LDR() NEXT
        IF GPXRB<31>=>
                (SUMM1@SUM<31:8>←-(GPXRB<31>@GPXRB<31:8>) NEXT
                GPXRB←SUM NEXT
                W1←0 NEXT
                NORMAL())
        END,

LDC:=
        BEGIN
        GPXRB←NOT MD
        END,

LAO:=
        BEGIN
        IF NOT MON=>LDR()
        END,
LMO:=
        BEGIN
        IF MON=>LDR()
        END,

STR:=
        BEGIN
        DECODE AMODE=>
                BEGIN
                0:=GPXRA←GPXRB,
    OTHERWISE:=MEM[MA]←GPXRB
                END
        END,

STE:=
        BEGIN
        DECODE AMODE=>
                BEGIN
                0:=GPXRA←EX,
        OTHERWISE:=MEM[MA]←EX
```

```
                END
        END,
!SW0-SW3 ARE NOT CODED HERE, BUT IF NEEDED THEY ARE LIKE STR, STE.

STD:=
        BEGIN
        DECODE AMODE=>
                BEGIN
                0:= (MA<=MEM[PC]<15:0> NEXT
                    INR←MEM[PC]),
    OTHERWISE:= (MEM[MA]←GPXRB NEXT
                    MEM[MA+4096]←GPXRB)
                END
        END,


STZ:=
        BEGIN
        DECODE AMODE=>
                BEGIN
                0:=GPXRA←0,
        OTHERWISE:=MEM[MA]←0
                END
        END,

SZD:=
        BEGIN
        DECODE AMODE=>
                BEGIN
                0:= STD(),  !NO STORING OF ZERO IF AMODE=0?
        OTHERWISE:= (MEM[MA]←0 NEXT
                    MEM[MA+4096]←0)
                END
        END,

STR2:=
        BEGIN
        DECODE AMODE=>
                BEGIN
                0:= (GPXRA←GPXRB NEXT
                    RA←RA+1 NEXT
                    RB←RB+1 NEXT
                    GPXRA←GPXRB),
        OTHERWISE:= (MEM[MA]←GPXRB NEXT
                    MA←MA+1 NEXT
                    RB←RB+1 NEXT
                    MEM[MA]←GPXRB)
                END
        END,

STR3:=
        BEGIN
        DECODE AMODE=>
```

```
                BEGIN
                0:= (GPXRA←GPXRB NEXT
                    RA←RA+1 NEXT
                    RB←RB+1 NEXT
                    STR2()),
        OTHERWISE:= (MEM[MA]←GPXRB NEXT
                    MA←MA+1 NEXT
                    RB←RB+1 NEXT
                    STR2())
                END
        END.

LOOP4:=                 !THIS IS ONLY CALLED WHEN AMODE=1
        BEGIN
        STD() NEXT
        RA←RA+1 NEXT
        RB←RB+1 NEXT
        MA←MA+1
        END.


STD2:=
        BEGIN
        DECODE AMODE=>
                BEGIN
                0:= STD(),
        OTHERWISE:= (LOOP4() NEXT
                    STD())
                END
        END.



STD3:=
        BEGIN
        DECODE AMODE=>
                BEGIN
                0:= STD(),
        OTHERWISE:= (LOOP4() NEXT
                    STD2())
                END
        END.

STD7:=
        BEGIN
        DECODE AMODE=>
                BEGIN
                0:= STD(),
        OTHERWISE:= (LOOP4() NEXT
                    LOOP4() NEXT
                    LOOP4() NEXT
                    LOOP4() NEXT
                    STD3())
```

```
                END
        END,

STH:=
        BEGIN
        MEM[MA]←MA SL0 16     !?
        END,

SPS:=
        BEGIN
        MEM[MA]←STATUS@PC<15:0>
                        !see p.66 of FTSC instruction set document
        END,

SPC:=
        BEGIN
        DECODE AMODE=>
                BEGIN
                0:= NO.OP(),
        OTHERWISE:=MEM[MA]←MEM[MA+4096]←STATUS@PC<15:0>
                END
        END,

!SBPA1:=
!       BEGIN
!       DECODE AMODE=>
!
!
!               1:= MEM[MA]←GPXRB NEXT
!
 !MAEC<7:0>  ARE MEMORY ADDRESS ERROR CODE BITS.  SEE 226 LINE 6 AND LAST
 !FOR  CONTRADICTORY INTERPRETATIONS.
!               END
!       END,

!SBPA0:=
!       BEGIN
!       DECODE AMODE=>
!               BEGIN
!               0:= GPXRA←GPXRB,
!               1:= MEM[MA]←GPXRB NEXT
!                   MAEC←0,
!               END
!       END,

!SBPD1:=
!       BEGIN
!       DECODE AMODE=>
!               BEGIN
!               0:= GPXRA←GPXRB,
!               1:= MEM[MA]←GPXRB NEXT
!                   MDEC<=1,
!!MDEC<7:0> IS MEMORY DATA ERROR CODE.
```

```
!               END
!       END,

!SBPD0:=
!       BEGIN
!       DECODE AMODE=>
!               BEGIN
!               0:=GPXRA←GPXRB,
!               1:= MEM[MA]←GPXRB NEXT
!                   MDEC←0,
!               END
!       END,


JMP:=
        BEGIN
        DECODE AMODE=>
                BEGIN
                0:=PC←GPXRA,
        OTHERWISE:=PC←MA
                END
        END,

JSB:=
        BEGIN
        GPXRB←STATUS@PC<15:0> NEXT
        DECODE AMODE=>
                BEGIN
                0:=PC←GPXRA,
        OTHERWISE:=PC←MA
                END
        END,

JPZ:=
        BEGIN
        IF NOT GPXRB<31>=>JMP()
        END,

JMI:=
        BEGIN
        IF GPXRB<31>=>JMP()
        END,

JZE:=
        BEGIN
        IF GPXRB EQL 0=>JMP()
        END,

JZEF:=
        BEGIN
        IF GPXRB<31:8> EQL 0=>JMP()
        END,
```

```
JNZ:=
        BEGIN
        IF GPXRB NEQ 0=>JMP()
        END,

JNZF:=
        BEGIN
        IF GPXRB<31:8> NEQ 0=>JMP()
        END,

JPS:=
        BEGIN
        IF GPXRB NEQ 0 AND GPXRB<31> EQL 0=>JMP()
        END,

JPSF:=
        BEGIN
        IF GPXRB<31:8> NEQ 0 AND GPXRB<31> EQL 0=>JMP()
        END,

JMZ:=
        BEGIN
        IF GPXRB<31> EQL 1 OR GPXRB EQL 0 =>JMP()
        END,

JMZF:=
        BEGIN
        IF GPXRB<31> EQL 1 OR GPXRB<31:8> EQL 0=>JMP()
        END,

JDN:=
        BEGIN
        SUMM1@SUM←(GPXRB<31>@GPXRB) -1 NEXT
        GPXRB←SUM NEXT
        IF GPXRB NEQ 0=>JMP()
        END,

JOS:=
        BEGIN
        IF OVFF=>JMP() NEXT
        OVFF←0
        END,

JCS:=
        BEGIN
        IF CRYFF=>JMP() NEXT
        CRYFF←0
        END,

OISN:=
        BEGIN
        MD←NOT MD NEXT
```

```
        IF (GPXRB OR MD) NEQ #37777777777 =>
        (PC←PC+1)
        END.

OISO:=
        BEGIN
        MD←NOT MD NEXT
        IF (GPXRB OR MD) EQL #37777777777 =>
        (PC←PC+1)
        END.

ASNZ:=
        BEGIN
        IF (GPXRB AND MD) NEQ 0=>PC←PC+1
        END.

ASZ:=
        BEGIN
        IF (GPXRB AND MD) EQL 0=>PC←PC+1
        END.


!CSNE AND CSEQ ARE NOT ON THE FLOWCHART DIAGRAMS

ADD:=
        BEGIN
        SUMM2@SUMM1@SUM←GPXRB<31>@GPXRB + MD<31>@MD NEXT
        IF SUMM2=>CRYFF←1 NEXT
        GPXRB←SUM NEXT
        IF (SUMM1 XOR SUM<31>)=>OVFF←1
        END.

SUB:=
        BEGIN
        SUMM2@SUMM1@SUM←GPXRB<31>@GPXRB + -(MD<31>@MD) NEXT
        IF SUMM2=>CRYFF←1 NEXT
        GPXRB←SUM NEXT
        IF (SUMM1 XOR SUM<31>)=>OVFF←1
        END.

MPY:=
        BEGIN
        EX←GPXRB NEXT
        GPXRB←0 NEXT
        COUNTER←0 NEXT
        CRYSTS←0 NEXT
        LOOP5:=
                REPEAT
                BEGIN
                ALUA<=GPXRB NEXT  !IT APPEARS THAT WE NEED GPXRB AND MD SIGN
                                  ! EXTENDED TWO BITS, SO HERE ALUA AND
                                  ! ALUB SHOULD BE 34 BITS.
                ALUB<=MD NEXT
```

```
                DECODE CRYSTS@EX<1:0>=>
                BEGIN
                        0:= (SUMM2@SUMM1@SUM←ALUA NEXT
                            CRYSTS←0),
                        1:= (SUMM2@SUMM1@SUM←ALUA+ALUB NEXT
                            CRYSTS←0),
                        2:= (SUMM2@SUMM1@SUM←ALUA-(ALUB SL0 1) NEXT
                            CRYSTS←1),
                        3:= (SUMM2@SUMM1@SUM←ALUA-ALUB NEXT
                            CRYSTS←1),
                        4:= (SUMM2@SUMM1@SUM←ALUA+ALUB NEXT
                            CRYSTS←0),
                        5:= (SUMM2@SUMM1@SUM←ALUA+(ALUB SL0 1) NEXT
                            CRYSTS←0),
                        6:= (SUMM2@SUMM1@SUM←ALUA-ALUB NEXT
                            CRYSTS←1),
                        7:= (SUMM2@SUMM1@SUM←ALUA NEXT
                            CRYSTS←1)
                END NEXT
                GPXRB←SUMM2@SUMM1@SUM<31:2> NEXT
                EX←SUM<1:0>@EX<31:2> NEXT
                COUNTER←COUNTER+1 NEXT
                IF COUNTER EQL 16=>LEAVE LOOP5
                END NEXT
!At this point in the computation, the sign appears in GPXRB<31:30>
!and the msb's to lsb's are in GPXRB<29:0>@EX.
        EX←EX<30:0>@GPXRB<31> NEXT
        GPXRB←GPXRB<30:0>@EX<31> NEXT
        W0←GPXRB NEXT  !IS THIS NEEDED FOR SOMETHING? FOR EXAMPLE
                        !IF MPY IS EXITED ON OVFF.
        IF GPXRB<31> XOR GPXRB<30>=>(OVFF←1 NEXT LEAVE MPY) NEXT
        GPXRB←EX SRR 1 NEXT  !EX ROTATED RIGHT ONE BIT
        EX←W0
        END,

        PPLOOP:=
                BEGIN
                REPEAT
                BEGIN
                COUNTER←COUNTER+1 NEXT
                DECODE INVERTOR=>(SUM←GPXRB-MD,SUM←GPXRB+MD) NEXT
                GPXRB←SUM<30:0>@EX<31> NEXT
                EX←EX<30:0>@NOT INVERTOR NEXT
                INVERTOR←SUM<31> NEXT
                IF COUNTER EQL 30=>LEAVE PPLOOP
                END
                END,


        PMLOOP:=
                BEGIN
                REPEAT
                BEGIN
```

```
                COUNTER←COUNTER+1 NEXT
                DECODE INVERTOR=>(SUM←GPXRB+MD,SUM←GPXRB-MD) NEXT
                GPXRB←SUM<30:0>@EX<31> NEXT
                EX←EX<30:0>@INVERTOR NEXT
                INVERTOR←SUM<31> NEXT
                IF COUNTER EQL 29=> LEAVE PMLOOP
                END
                END,


DIVPP:=
        BEGIN
        EX←EX SL0 1 NEXT
        SUM←GPXRB-MD NEXT  !IN DIV GPXRB=0 HERE.
        IF NOT SUM<31>=>(DIVFF←1 NEXT LEAVE DIVPP) NEXT !DENOM=0 IN DIV
                !AND MSB HALF OF NUMERATOR GEQ DENOMINATOR IN LDV
        GPXRB←SUM<30:0>@EX<31> NEXT
        EX←EX SL0 1 NEXT
        INVERTOR←1 NEXT
        COUNTER←0 NEXT
        PPLOOP() NEXT
        DECODE INVERTOR=>(SUM←GPXRB-MD,SUM←GPXRB+MD) NEXT
        GPXRB←SUM NEXT
        EX←EX<30:0>@NOT INVERTOR NEXT
        DECODE GPXRB<31>=>
                BEGIN
                0:= (W0←GPXRB NEXT
                        GPXRB←(EX SL0 1) + 1 NEXT
                        EX←W0),
                1:= (W0←GPXRB + MD NEXT
                        GPXRB←EX SL0 1 NEXT
                        EX←W0)
                END
        END,


DIVPM:=
        BEGIN
        !THE STEP EX←EX SL0 1 IS TAKEN CARE OF A FEW LINES HENCE. ,
        SUM←GPXRB+MD NEXT !IN DIV GPXRB=0 HERE.
        IF SUM GTR 0=>(DIVFF←1 NEXT LEAVE DIVPM) NEXT
        !THIS IS IMPOSSIBLE FOR DIV. FOR LDV THIS CHECKS IF MS HALF
        !OF NUM GTR  ABSOLUTE VALUE OF DENOMINATOR.
        IF (SUM EQL 0) AND (W0 NEQ 0)=>(DIVFF←1 NEXT LEAVE DIVPM) NEXT
        !THIS IS ALSO IMPOSSIBLE FOR DIV. IN LDV IT CHECKS IF MS HALF
        !OF NUMERATOR EQL ABSOLUTE VALUE OF DENOMINATOR AND LS HALF
        !OF NUMERATOR NEQ 0.
        GPXRB←SUM<30:0>@EX<30> NEXT
        EX←EX<29:0>@'01 NEXT
        IF SUM EQL 0=>INVERTOR←0 NEXT  !AT THIS POINT LS HALF OF
                !NUM IN LDV IS KNOWN TO BE ZERO.
        IF SUM LSS 0=>(
        SUM←GPXRB-MD NEXT
```

```
        GPXRB←SUM<30:0>@EX<31> NEXT
        EX←EX SL1 1 NEXT
        INVERTOR←SUM<31>' NEXT
        COUNTER←0 NEXT
        PMLOOP() NEXT
        DECODE INVERTOR=>(GPXRB←GPXRB+MD,GPXRB←GPXRB-MD) NEXT
        EX←EX<30:0>@INVERTOR NEXT
        DECODE GPXRB<31>=>((W0←GPXRB NEXT EX←EX SL0 1),
                           (W0←GPXRB-MD NEXT EX←EX SL1 1)) NEXT
        GPXRB←EX+1 NEXT
        EX←W0
        END,

DIVMP:=
        BEGIN
        EX←EX SL0 1 NEXT
        GPXRB←GPXRB+MD  NEXT !HERE IN DIV GPXRB←MD-1.
        IF GPXRB<31>=>(DIVFF←1 NEXT LEAVE DIVMP) NEXT !CHECK FOR DENOM=0
! ACTUALLY WE MAY HAVE TO EXECUTE ALSO 646 IN ORDER THAT THE PROPER
! VALUES BE IN GPXRB AND EX WHEN CONTROL GETS THE INTERRUPT SIGNAL.
        GPXRB←GPXRB<30:0>@EX<31> NEXT
        EX←EX SL1 1 NEXT
        SUM←GPXRB-MD NEXT
        INVERTOR←SUM<31> NEXT
        GPXRB←SUM<30:0>@EX<31> NEXT
        EX←EX SL1 1 NEXT
        PPLOOP() NEXT   !SAME LOOP AS FOR +/+
        DECODE INVERTOR=>(SUM←GPXRB-MD,SUM←GPXRB+MD) NEXT
        GPXRB←SUM NEXT
        EX←EX<30:0>@NOT INVERTOR NEXT
        IF GPXRB GTR 0=>(W0←GPXRB-MD NEXT  !E.G.-4/5
                         EX←EX SL1 1 NEXT
                         GPXRB←EX + 1 NEXT
                         EX←W0 NEXT
                         LEAVE DIVMP) NEXT
        IF GPXRB EQL 0=>(GPXRB←(EX SL0 1) + 1 NEXT !E.G.-4/4
                         EX←0 NEXT LEAVE DIVMP) NEXT
        IF GPXRB LSS 0=>(GPXRB←GPXRB+MD NEXT
                         IF GPXRB EQL 0=>(GPXRB←EX SL0 1 NEXT !E.G.-4/2
                                          EX←0 NEXT LEAVE DIVMP) NEXT
                         IF GPXRB NEQ 0=>(W0←GPXRB-MD NEXT !E.G.-4/3
                                          EX←EX SL0 1 NEXT
                                          GPXRB←EX + 1 NEXT
                                          EX←W0))
        END,


DIVMM:=
        BEGIN
        EX←EX SL0 1 NEXT
        SUM←GPXRB-MD NEXT !IN DIV GPXRB=-1 HERE.
        IF SUM LSS 0=>(DIVFF←1 NEXT LEAVE DIVMM) NEXT
        !THIS IS IMPOSSIBLE FOR DIV.  IN LDV THIS CHECKS IF MD GTR GPXRB
```

```
        !I.E., IF ABS.VALUE OF MS HALF OF NUMERATOR GTR ABS. VALUE OF
        !DENOMINATOR.
        GPXRB←SUM<30:0>@EX<31> NEXT
        EX←EX SL0 1 NEXT
        IF (SUM EQL 0) AND (EX EQL 0)=>(DIVFF←1 NEXT LEAVE DIVMM) NEXT
!SO '1X0...0/-1 YIELDS DIVFF IN DIV SINCE SUM EQL 0<=> MD=-1 IN DIV.
!IN LDV THIS CHECKS IF MS HALF OF NUM = DENOMINATOR AND LS HALF =
!'1X0...0 AS IN DIV. AGAIN THIS DOES NOT MAKE SENSE.
        COUNTER←0 NEXT
        INVERTOR←0 NEXT
        PMLOOP() NEXT !SAME LOOP AS +/-.
        DECODE INVERTOR=>(GPXRB←GPXRB+MD,GPXRB←GPXRB-MD) NEXT
        EX←EX<30:0>@INVERTOR NEXT
        IF GPXRB EQL 0=>(GPXRB←(EX SL0 1) + 1 NEXT !E.G.-4/-4
                         EX←0 NEXT LEAVE DIVMM) NEXT
        IF GPXRB GTR 0=>(W0←GPXRB+MD NEXT !E.G.-4/-5
                         GPXRB←EX SL0 1 NEXT
                         EX←W0 NEXT LEAVE DIVMM) NEXT
        IF GPXRB LSS 0=>(W0←GPXRB-MD NEXT
                         IF W0 EQL 0=>(EX←EX SL1 1 NEXT !E.G.-4/-2
                                       GPXRB←EX + 1 NEXT
                                       EX←0 NEXT LEAVE DIVMM) NEXT
                         IF W0 NEQ 0=>(W0←GPXRB+MD NEXT !E.G.-4/-3
                                       EX←EX SL0 1 NEXT
                                       GPXRB←EX+1 NEXT
                                       EX←W0))
        END.

DIVI:=  !GPXRB CONTAINS NUMERATOR AND MD DENOMINATOR.  QUOTIENT GOES IN
!               GPXRB WITH REMAINDER IN EX. SIGN OF REMAINDER IS SAME AS
!               SIGN OF NUMERATOR.
        BEGIN
        EX←GPXRB NEXT
        DECODE GPXRB<31>@MD<31>=>
                BEGIN
                0:=(GPXRB←0 NEXT DIVPP()),  !+/+
                1:=(GPXRB←0 NEXT DIVPM()), !+/-
                2:=(GPXRB←-1 NEXT DIVMP()), !-/+
                3:=(GPXRB←-1 NEXT DIVMM())  !-/-
                END
        END.

LDV:= !NUMERATOR IS EX@GPXRB, DENOMINATOR IS MD. (PROBABLY) EX<31>=GPXRB<31>.
        !OTHER DETAILS AS IN DIV.
        !(AT THE START EX AND GPXRB ARE INTERCHANGED)
        BEGIN
        W0←GPXRB NEXT
        GPXRB←EX NEXT
        EX←W0 NEXT
        W0←EX SL0 1 NEXT
        IF (GPXRB GEQ 0) AND (MD GEQ 0)=>(DIVPP() NEXT LEAVE LDV) NEXT
```

```
        IF (GPXRB GEQ 0) AND (MD LSS 0)=>(DIVPM() NEXT LEAVE LDV) NEXT
        IF (GPXRB LSS 0) AND (MD GEQ 0)=>(DIVMP() NEXT LEAVE LDV) NEXT
        IF (GPXRB LSS 0) AND (MD LSS 0)=>(DIVMM() NEXT LEAVE LDV)
        END,

ACO:=
        BEGIN
        SUMM2@SUMM1@SUM←(GPXRB<31>@GPXRB)+(MD<31>@MD)+CRYFF NEXT
        GPXRB←SUM NEXT
        IF SUMM2=>CRYFF←1 NEXT
        IF SUMM1 XOR SUM<31>=>OVFF←1
        END,

LOOP6:=
        BEGIN
        REPEAT
                BEGIN
                GPXRB←GPXRB SL0 2 NEXT

                W1←W1-2 NEXT
                IF NOT((GPXRB NEQ 0 AND GPXRB<31:29> EQL 0) OR
                        (GPXRB <31:29> EQL 7))=> LEAVE LOOP6
                END
        END,


CFL:=     !TAKES GPXRA OR MD INTEGER AND CONVERTS TO FLOATING IN GPXRB.
! THE MANTISSA IS INTERPRETED AS A BINARY FRACTION LESS THAN 1.
        BEGIN
        GPXRB←MD NEXT
        W1←0 NEXT
        IF (GPXRB NEQ 0) AND (GPXRB<31:29> EQL 0 OR GPXRB<31:29> EQL 7)=>
        LOOP6() NEXT
!LOOP6 CAN CHANGE THE VALUE OF W1.
        IF GPXRB EQL 0=>(GPXRB←"80 NEXT LEAVE CFL) NEXT
        IF GPXRB<31> XOR GPXRB<30>=>
!ALREADY NORMALIZED
        (GPXRB<7:0>←W1<7:0> NEXT GPXRB<7:0>←GPXRB<7:0>+31 NEXT LEAVE CFL) NEXT
        IF (GPXRB<31:29> EQL 6) OR (GPXRB<31:29> EQL 1) =>
!NORMALIZE FIRST
        (GPXRB←GPXRB SL0 1 NEXT GPXRB<7:0>←W1-1 NEXT
                                GPXRB<7:0>←GPXRB<7:0>+31)
        END,


ADDLP1:=
        BEGIN
        REPEAT
                BEGIN
                GPXRB<31:8>←GPXRB<31>@GPXRB<31>@GPXRB<31:10> NEXT  !?
                W1<7:0>←W1<7:0>+2 NEXT
                GPXRB<7:0>←MD<7:0> NEXT ,
                IF W1<7:0> EQL 0=>LEAVE ADDLP1
```

```
                END
        END,

ADDLP2:=
        BEGIN
        REPEAT
                BEGIN
                W1<7:0>←W1<7:0>-2 NEXT
                MD<31:8>←MD<31>@MD<31>@MD<31:10> NEXT
                IF W1<7:0> EQL 0=>LEAVE ADDLP2
                END
        END,


ADDF:=
        BEGIN

!FIRST COMPARE THE TWO EXPONENTS.  THEIR DIFFERENCE GOES IN EXPOUT.

        EXPOUT←GPXRB<7>@GPXRB<7:0>-MD<7>@MD<7:0> NEXT
        W1<7:0>←SUM<7:0>←EXPOUT<7:0> NEXT
! WHAT DO WE NEED W1 FOR?
        IF SUM<7:0> EQL 0=>(DECODE ASUBE=>(GPXRB<31:8>←GPXRB<31:8>+MD<31:8>,
                                        GPXRB<31:8>←GPXRB<31:8>-MD<31:8>) NEXT
                        W1←0 NEXT NORMAL() NEXT LEAVE ADDF) NEXT
! SO FROM HERE ON SUM<7:0> NEQ 0.
        OVF8←EXPOUT<8> XOR EXPOUT<7> NEXT
        EXG23←(EXPOUT<7:0> GTR 23) OR (EXPOUT<7:0> LSS -23) NEXT
        DECODE EXG23@OVF8@SUM<0>@SUM<7>=>
                BEGIN
                [4:7,#14:#17]:=((IF GPXRB<7>=>  !IF GPXRB EXPONENT <0
                                (DECODE ASUBE=>(GPXRB←MD,
                                        GPXRB←(-MD<31:8>)@MD<7:0>))) NEXT
                                LEAVE ADDF),
                    [#11,#13]:=(DECODE ASUBE=>(GPXRB←MD,
                                GPXRB←(-MD<31:8>)@MD<7:0>) NEXT
                                LEAVE ADDF),
                    [#10,#12]:=LEAVE ADDF,
                            1:= ADDLP1(),


                            3:= (GPXRB<31:8>←GPXRB<31>@GPXRB<31:9> NEXT
                                W1<7:0>←W1<7:0>+1 NEXT
                                GPXRB<7:0>←MD<7:0> NEXT
                                ADDLP1()),

                            0:= ADDLP2(),
                            2:=( W1<7:0>←W1<7:0>-1 NEXT
                                MD<31:8>←MD<31>@MD<31:9> NEXT
                                ADDLP2())
                END NEXT
```

```
                                        ALUA<=GPXRB NEXT !SIGN EXTEND ONE BIT
                                        ALUB<=MD NEXT
                                        DECODE ASUBE=>
                                        BEGIN
                                        0:=SUMM2@SUMM1@GPXRB<31:8>←
                                           GPXRB<31>@GPXRB<31:8>+MD<31>@MD<31:8>,
                                        1:=SUMM2@SUMM1@GPXRB<31:8>←
                                           GPXRB<31>@GPXRB<31:8>-MD<31>@MD<31:8>
                                        END NEXT
                                        IF SUMM2=>CRYFF←1
        END,



SUBF:=
        BEGIN
        ASUBE←1 NEXT  !AUTO SUBTRACT ENABLE
        ADDF()
        END,


!From here to the end of MPYF has been checked in MPYFML


!Version of Mar.3,1978
MINUS1:=
        BEGIN
                                        GPXRB<31:8>←GPXRB<30:8>@EX<31> NEXT
                                        EX←EX SL0 1 NEXT
                                        EXPA9<=GPXRB<7:0> NEXT
                                        EXPOUT←EXPA9-1 NEXT
                                        OVF8←EXPOUT<8> XOR EXPOUT<7> NEXT
                                        GPXRB<7:0>←EXPOUT<7:0>
                                        END,

MINUS2:=
        BEGIN
                                        GPXRB<31:8>←GPXRB<29:8>@EX<31:30> NEXT
                                        EX←EX SL0 2 NEXT
                                        EXPA9<=GPXRB<7:0> NEXT
                                        EXPOUT←EXPA9-2 NEXT
                                        OVF8←EXPOUT<8> XOR EXPOUT<7> NEXT
                                        GPXRB<7:0>←EXPOUT<7:0>
                                        END,


MPYF1A:=
                BEGIN
                                MLOOP1:=
                                    REPEAT
```

```
                                BEGIN
                                IF (OVF8@GPXRB<31:29> NEQ 0 AND
                                    OVF8@GPXRB<31:29> NEQ 7)=>
                                    LEAVE MLOOP1 NEXT
                                MINUS2()
                                END   !Of MLOOP1
                                NEXT
                                IF (OVF8@GPXRB<31:29> EQL 6 OR
                                    OVF8@GPXRB<31:29> EQL 1)=>
                                        MINUS1() NEXT
                                EX<31:8>←'0@EX<31:9> NEXT
                                IF OVF8=>OVFF←1
                END,

MPYF1B:=
        BEGIN
                REPEAT
                BEGIN
                DECODE OVF8@GPXRB<31:29>=>
                        BEGIN
                        [1,6]:=MINUS1(),
                        [0,7]:=MINUS2(),
                    [#11,#16]:=(MINUS1() NEXT
                                EX<31:8>←'0@EX<31:9> NEXT
                                IF OVF8=>OVFF←1 NEXT LEAVE MPYF1B),
                        [2:5]:=(OVFF←1 NEXT LEAVE MPYF1B),
                    [#12:#15]:=(EX<31:8>←'0@EX<31:9> NEXT LEAVE MPYF1B),
                    [#10,#17]:=(MINUS2() NEXT
                                MPYF1A() NEXT LEAVE MPYF1B)
                        END   !of decode
                END           !oREPEAT
        END,                  !of MPYF1B


MPYF:=   !TAKES NORMALIZED GPXRB AND MD (OR GPXRA) IN FLOATING POINT
         !FORM AND PUTS THE SIGN OF THE PRODUCT AND 23 MSB'S IN
         !MANTISSA OF GPXRB, 23 LSB'S IN BITS <30:8> OF EXTENSION
         !REGISTER, AND EXPONENT IN EXPONENT OF GPXRB.

        BEGIN
        EX←GPXRB<31:8>@'00000000 NEXT
        GPXRB<31:8>←0 NEXT
        IF NOT AMODE=>(MD←GPXRA) NEXT  !OTHERWISE USE OLD MD
        COUNTER←0 NEXT
        CRYSTS←0 NEXT
        LOOPSF:=

                REPEAT

                BEGIN
                ALUA<33:8><=GPXRB<31:8> NEXT  !HERE ALSO(AS IN MPY INTEGER)
                                              !SIGN-EXTEND TWO BITS)
                ALUB<33:8><=MD<31:8> NEXT
```

```
                DECODE CRYSTS@EX<9:8>=>
                BEGIN
                        0:= (SUMM2@SUMM1@SUM<31:8>←ALUA<33:8> NEXT
                            CRYSTS←0),
                        1:= (SUMM2@SUMM1@SUM<31:8>←ALUA<33:8>+ALUB<33:8> NEXT
                            CRYSTS←0),
                        2:= (SUMM2@SUMM1@SUM<31:8>←
                             ALUA<33:8>-(ALUB<33:8> SL0 1) NEXT
                            CRYSTS←1),
                        3:= (SUMM2@SUMM1@SUM<31:8>←ALUA<33:8>-ALUB<33:8> NEXT
                            CRYSTS←1),
                        4:= (SUMM2@SUMM1@SUM<31:8>←ALUA<33:8>+ALUB<33:8> NEXT
                            CRYSTS←0),
                        5:= (SUMM2@SUMM1@SUM<31:8>←
                             ALUA<33:8>+(ALUB<33:8> SL0 1) NEXT
                            CRYSTS←0),
                        6:= (SUMM2@SUMM1@SUM<31:8>←ALUA<33:8>-ALUB<33:8> NEXT
                            CRYSTS←1),
                        7:= (SUMM2@SUMM1@SUM<31:8>←ALUA<33:8> NEXT
                            CRYSTS←1)
                END NEXT
                GPXRB<31:8>←SUMM2@SUMM1@SUM<31:10> NEXT
                EX<31:8>←SUM<9:8>@EX<31:10> NEXT
                COUNTER←COUNTER+1 NEXT
                IF COUNTER EQL 12=>LEAVE LOOP5F
                END NEXT
        IF GPXRB<31:8> EQL 0=>(GPXRB←"80 NEXT EX←0 NEXT LEAVE MPYF) NEXT
                ! THAT IS THE FLOATING REPRESENTATION OF ZERO.
!Eliminate GPSDECODE                            GPSDECODE() NEXT LEAVE MPYF) NEXT
        EXPA9<=GPXRB<7:0> NEXT    !HERE WE ARE USING THE MICROMACHINE NOTATION.
        EXPB9<=MD<7:0> NEXT
        EXPOUT←EXPA9+EXPB9 NEXT
        GPXRB<7:0>←EXPOUT<7:0> NEXT
        OVF8←EXPOUT<8> XOR EXPOUT<7> NEXT
        DECODE GPXRB<31:29>=>
                BEGIN
                [0,7]:=(GPXRB<31:8>←GPXRB<29:8>@EX<31:30> NEXT
                        EX←EX SL0 2 NEXT
                        DECODE OVF8=>
!We do the decode on OVF8 here , even though the first 4 lines
!in both cases are the same, because OVF8 is recalculated immediately
!and I didn't see any other way that would not introduce fictitious
!registers or something like the DELAY.
                        BEGIN
                        0:= (EXPA9<=GPXRB<7:0> NEXT
                                EXPOUT←EXPA9-1 NEXT
                                OVF8←EXPOUT<8> XOR EXPOUT<7> NEXT
                                GPXRB<7:0>←EXPOUT<7:0> NEXT
                                MPYF1A()),
                        1:= (EXPA9<=GPXRB<7:0> NEXT
                                EXPOUT←EXPA9-1 NEXT
                                OVF8←EXPOUT<8> XOR EXPOUT<7> NEXT
                                GPXRB<7:0>←EXPOUT<7:0>  NEXT
```

```
                        MPYF1B())
                END),
        [1,6]:= (GPXRB<31:8>←GPXRB<30:8>⊕EX<31> NEXT
                EX←EX SL0 1 NEXT
                EX<31:8>←'0⊕EX<31:9> NEXT
                IF OVF8=>OVFF←1),

        2:= (DECODE OVF8=>
                BEGIN
                0:=(EXPA9<=GPXRB<7:0> NEXT
                        EXPOUT←EXPA9+1 NEXT
                        OVF8←EXPOUT<8> XOR EXPOUT<7> NEXT
                        GPXRB<7:0>←EXPOUT<7:0> NEXT
                        EX<31:8>←'0⊕EX<31:9>  NEXT
                        IF OVF8=>OVFF←1),
                1:=(EXPA9<=GPXRB<7:0> NEXT
                        EXPOUT←EXPA9+1 NEXT
                        OVF8←EXPOUT<8> XOR EXPOUT<7> NEXT
                        GPXRB<7:0>←EXPOUT<7:0> NEXT
                        EX<31:8>←'0⊕EX<31:9> NEXT
                        IF NOT OVF8=>OVFF←1)  !here's the difference

                END)
        END
        !WHY ARE 3 AND 5 IMPOSSIBLE?  Because GPXRB<31:30> containe
        !the sign bit repeated.  2 is only possible for -2↑23x-2↑23.
END,



!  From the last landmark to here has been checked as MPYFML


!  From here to the end of DIVF has been checked along with NORMAL,Mar.8,78
! as DIVFML.

FPPLOOP:=
        BEGIN
        REPEAT
                BEGIN
                COUNTER←COUNTER+1 NEXT
                DECODE INVERTOR=>(SUM<31:8>←GPXRB<31:8>-MD<31:8>,
                                SUM<31:8>←GPXRB<31:8>+MD<31:8>) NEXT
                EX<31:8>←EX<30:8>⊕NOT INVERTOR NEXT
                INVERTOR←SUM<31> NEXT

                IF COUNTER EQL 23=> LEAVE FPPLOOP NEXT
                GPXRB<31:8>←SUM<30:8>⊕EX<31>
                END
        END,
```

```
FPMLOOP:=
        BEGIN
        REPEAT
                BEGIN
                COUNTER←COUNTER+1 NEXT
                DECODE INVERTOR=>(SUM<31:8>←GPXRB<31:8>+MD<31:8>,
                                  SUM<31:8>←GPXRB<31:8>-MD<31:8>) NEXT
                EX<31:8>←EX<30:8>@ INVERTOR NEXT
                INVERTOR←SUM<31> NEXT
                IF COUNTER EQL 23=> LEAVE FPMLOOP NEXT
                GPXRB<31:8>←SUM<30:8>@EX<31>
                END
        END,
DIVF:=    !GPXRB/MD
        BEGIN
        IF NOT AMODE=>MD←GPXRA NEXT
        W1<7:0>←"FF NEXT    !??
        EX<31>←GPXRB<8> NEXT
        GPXRB<31:8>←GPXRB<31>@GPXRB<31:9> NEXT
        IF MD EQL 0=>(DIVFF←1 NEXT LEAVE DIVF) NEXT
        DECODE GPXRB<31>@MD<31>=>
                BEGIN
                0:=(COUNTER←0 NEXT
                        SUM<31:8>←GPXRB<31:8>-MD<31:8> NEXT
                        GPXRB<31:8>←SUM<30:8>@EX<31> NEXT
                        EX<31:8>←EX<30:8>@'0 NEXT
                        INVERTOR←SUM<31> NEXT
                        FPPLOOP() NEXT
                        GPXRB<31:8>←SUM<31:8> NEXT
                        W1<31:8>←0 NEXT !FOR USE IN NORMAL() ??

                        DECODE GPXRB<31>=>
                                BEGIN
                                0:=(W0<31:8>←GPXRB<31:8> NEXT
                                        GPXRB<31:8>←EX<31:8> SL0 1 + 1 NEXT
                                        EX<31:8>←W0<30:8>@'1), !?
                                1:=(W0<31:8>←GPXRB<31:8>+MD<31:8> NEXT
                                        GPXRB<31:8>←EX<31:8> SL0 1 NEXT
                                        EX<31:8>←W0 SL0 1)
                                END),

                1:=(COUNTER←0 NEXT
                        SUM<31:8>←GPXRB<31:8>+MD<31:8> NEXT
                        GPXRB<31:8>←SUM<30:8>@EX<31> NEXT
                        EX<31:8>←EX<30:8>@'1 NEXT
                        INVERTOR←SUM<31> NEXT
                        FPMLOOP() NEXT
                        GPXRB<31:8>←SUM<31:8> NEXT
                        W1<31:8>←0 NEXT
                        DECODE GPXRB<31>=>
                                BEGIN
                                0:=(W0<31:8>←GPXRB<31:8> NEXT
```

```
                                GPXRB<31:8>←EX<31:8> SL0 1 + 1 NEXT
                                EX<31:8>←W0<30:8>@'1),!?
                        1:=(W0<31:8>←GPXRB<31:8>-MD<31:8> NEXT
                                GPXRB<31:8>←EX<31:8> SL1 1 + 1 NEXT
                                EX<31:8>←W0<30:8>@'1) !?
                        END),


        2:=(COUNTER←0 NEXT
                SUM<31:8>←GPXRB<31:8>+MD<31:8> NEXT
                GPXRB<31:8>←SUM<30:8>@EX<31> NEXT
                EX<31:8>←EX<30:8>@'1 NEXT
                INVERTOR←SUM<31> NEXT
                FPPLOOP() NEXT
                GPXRB<31:8>←SUM<31:8> NEXT
                W1<31:8>←0 NEXT
                DECODE GPXRB<31>=>
                        BEGIN
                        0:=(W0<31:8>←GPXRB<31:8>-MD<31:8> NEXT
                                GPXRB<31:8>←EX<31:8> SL1 1 +1 NEXT
                                EX<31:8>←W0<30:8>@'1), !?
                        1:=(GPXRB<31:8>←EX<31:8> SL0 1 + 1 NEXT
                                EX<31:8>←W0<30:8>@'1) !?
                        END),

        3:=(COUNTER←0 NEXT
                SUM<31:8>←GPXRB<31:8>-MD<31:8> NEXT
                GPXRB<31:8>←SUM<30:8>@EX<31> NEXT
                EX<31:8>←EX<30:8>@'0 NEXT
                INVERTOR←SUM<31> NEXT
                FPMLOOP() NEXT
                GPXRB<31:8>←SUM<31:8> NEXT
                W1<31:8>←0 NEXT
                DECODE GPXRB<31>=>
                        BEGIN
                        0:=(W0<31:8>←GPXRB<31:8>+MD<31:8> NEXT
                                GPXRB<31:8>←EX<31:8> SL0 1 NEXT
                                EX<31:8>←W0<31:8> SL0 1),
                        1:=(GPXRB<31:8>←EX<31:8> SL0 1 +1 NEXT
                                EX<31:8>←W0<30:8>@'1) !?
                        END)
        END
        NEXT
        SUMM1@SUM<31:8>←GPXRB<31>@GPXRB<31:8> +
                        GPXRB<31>@GPXRB<31:8> NEXT
        GPXRB<31:8>←SUM<31:8> NEXT
        EXPOUT←GPXRB<7>@GPXRB<7:0> - MD<7>@MD<7:0> NEXT
        OVF8←EXPOUT<8> XOR EXPOUT<7> NEXT
        GPXRB<7:0>←EXPOUT<7:0> NEXT
        NORMAL()
END,

!From FPPLOOP to here has been checked along with NORMAL.
```

```
!From here to the end of SRTF has been checked as SQRTML

SRTF:=    !TAKES FLOATING GPXRA OR MD AND PUTS SQUARE ROOT IN GPXRB.
          !A = m x 2↑e.                    (↑ indicates exponentiation)

        BEGIN
        IF NOT AMODE=>(MD←GPXRA) NEXT
        IF MD LSS 0=>(OVFF←1 NEXT LEAVE SRTF) NEXT
        IF MD EQL 0=>(GPXRB←"80 NEXT LEAVE SRTF) NEXT
        W1←MD NEXT  !This register transfer is pure machine dependence.
        W0<31:8>←MD<31:8> SL0 1 NEXT  !This too.
        !W0<31:9> = M = m x 2↑23.
        W0<7:0>←0 NEXT
        DECODE MD<0>=>   !Even or odd exponent
                BEGIN
                0:=(GPXRB←W0<31:30> NEXT  !If even, shift argument two bits
                   W0←W0 SL0 2) ,         !left into GPXRB.
                        ! GPXRB@W0<31:10>=2M,  W1=e.
                1:=(GPXRB←W0<31> NEXT  !If odd, shift argument one bit left
                   W0←W0 SL0 1 NEXT   !into GPXRB and add 1 to exponent.
                   EXPOUT←MD<7>@MD<7:0> + 1 NEXT  !Exponent overflow check.
                   W1<7:0>←EXPOUT<7:0>)
                        ! GPXRB@W0<31:10> = M, W1=e+1.
                END
        ! In any case, at this stage sqrt(GPXRB@W0<31:10>) x 2↑(W1/2) equals
        ! 2↑12 x sqrt(m) x 2↑(e/2).  So all we have to do is take W1/2 for
        !the exponent of the answer and for the mantissa (in its fractional
        !form) take sqrt(GPXRB@W0<31:10>) x 2↑(-12).  What is the same is
        !to take sqrt(GPXRB@W0<31:10> x 2↑22) x 2↑(-23), in other words,
        !take the square root mentioned here as an integer, and then just
        !interpret it as a fraction in the bits <30:8> of the register
        !containing the answer (bit <31> will be zero, since we are finding
        !the positive square root).  Notice that GPXRB<1:0>@W0<31:10> x 2↑22 is
        !a 46-bit number, and the range of values is >=2↑44 and <=2↑46 - 2↑22.

        NEXT
        W1<7:0>←W1<7>@W1<7:1> NEXT  !Exponent of root is 1/2 previous value.
        IF EXPOUT<8> XOR EXPOUT<7>=>W1<7:0>←#100 NEXT
                !This is 1/2 of previous value
                                                !in the case of overflow.
        !So from here to the end we will be finding the square root of the
        !integer GPXRB<1:0>@W0<31:10> x 2↑22.
        !For proof of the following, see <MARCUS>SRTFPROOF.XOF.
        COUNTER←0  NEXT
        W1<31:8>←0 NEXT  !Zero is partial square root.
        SUM←GPXRB-1 NEXT
        GPXRB←SUM<29:0>@W0<31:30> NEXT
```

```
        SLOOP:=
                REPEAT
                BEGIN
                W0←W0 SL0 2 NEXT
                COUNTER←COUNTER+1 NEXT
                DECODE SUM<31>=>        !positive (or zero) or neg remainder
                BEGIN
                0:= (W1<31:8>←2*W1<31:8> + 1 NEXT
                        IF COUNTER EQL 23=>(LEAVE SLOOP) NEXT
                        W2←4*W1<31:8>  + 1 NEXT
                        SUM←GPXRB-W2 NEXT
                        GPXRB←SUM<29:0>@W0<31:30>),
                1:= (W1<31:8>←2*W1<31:8>  NEXT
                        IF COUNTER EQL 23=>(LEAVE SLOOP) NEXT
                        W2←4*W1<31:8> + 3 NEXT
                        SUM←GPXRB+W2 NEXT
                        GPXRB←SUM<29:0>@W0<31:30>)

                END
                END
        NEXT
        GPXRB←W1
        END.




VADDF:=
        BEGIN
        ADDF() NEXT
        RA←RA+1 NEXT
        RB←RB+1 NEXT
        MA←MA+1 NEXT
        IF AMODE=>MD←MEM[MA] NEXT
        ADDF() NEXT
        RA←RA+1 NEXT
        RB←RB+1 NEXT
        MA←MA+1 NEXT
        IF AMODE=>MD←MEM[MA] NEXT
        ADDF()
        END.


        !VSUBF IS NOT ON FLOW DIAGRAMS

VMPYF:=
        BEGIN
        MPYF() NEXT
```

```
        RA←RA+1 NEXT
        RB←RB+1 NEXT
        MA←MA+1 NEXT
        IF AMODE=>MD←MEM[MA] NEXT
        MPYF() NEXT
        RA←RA+1 NEXT
        RB←RB+1 NEXT
        MA←MA+1 NEXT
        IF AMODE=>MD←MEM[MA] NEXT
        MPYF()
        END,

VIPF:=
        BEGIN
        MPYF() NEXT
        W3←GPXRB NEXT
        MA←MA+1 NEXT
        RA←RA+1 NEXT
        RB←RB+1 NEXT
        IF AMODE=>MD←MEM[MA] NEXT
        MPYF() NEXT
        W2←GPXRB NEXT
        MA←MA+1 NEXT
        RB←RB+1 NEXT
        RA←RA+1 NEXT
        DECODE AMODE=>
                BEGIN
                0:=W0←GPXRA,
        OTHERWISE:=MD←MEM[MA]
                END
        NEXT
        MPYF() NEXT
        DECODE AMODE=>
                BEGIN
                0:=GPXRA←W2,
                1:=MD←W2
                END
        NEXT
        ADDF() NEXT
        DECODE AMODE=>
                BEGIN
                0:= (GPXRA←W2 NEXT
                    ADDF() NEXT
                    GPXRA←W0),
        OTHERWISE:= (MD←W3 NEXT
                    ADDF())
                END
        END,

VSMF:=
        BEGIN
        MPYF() NEXT
        RB←RB+1 NEXT
```

```
        MPYF() NEXT
        RB←RB+1 NEXT
        MPYF()
        END,

LOOP7:=
        BEGIN
        REPEAT
                BEGIN
                W1←W1+2 NEXT
                GPXRB←GPXRB<31>@GPXRB<31>@GPXRB<31:2> NEXT
                IF W1<7:0> EQL 0=> LEAVE LOOP7
                END
        END,


CFX:=   !CONVERTS GPXRA OR MD IN FLOATING TO GPXRB INTEGER.
        BEGIN
        GPXRB←MD NEXT
        IF GPXRB<7:0> EQL 0 =>          !EXPONENT=0
                (GPXRB←GPXRB SL0 1 NEXT
                (IF GPXRB EQL 0 => GPXRB<=1 NEXT LEAVE CFX) NEXT
                (IF GPXRB NEQ 0 => GPXRB←0) NEXT LEAVE CFX) NEXT
        ! CONVERTS <X00...0> TO -1 AND A NON-ZERO NUMBER OF ABSOLUTE
        ! VALUE < 1 TO 0.
        IF GPXRB<7>=>(GPXRB←0 NEXT LEAVE CFX) NEXT  !NEG EXPONENT GOES TO 0.
        IF GPXRB<7:0> NEQ 0 AND GPXRB<7> EQL 0 =>   !POSITIVE EXPONENT
        (W1←31 NEXT
        W1←GPXRB<7:0>-W1<7:0> NEXT
        GPXRB<7:0>←0 NEXT
        DECODE W1<7>@W1<0>=>
                BEGIN
                0:= IF W1<7:0> NEQ 0 =>OVFF←1,
                        !IF W1<7:0> EQL 0, JUST LEAVE CFX.
                1:= OVFF←1,
                2:= LOOP7(),
                3:= (W1←W1+1 NEXT
                    GPXRB←GPXRB<31>@GPXRB<31:1> NEXT
                    IF W1 NEQ 0=>LOOP7())
                END)
        END,


UPF:=    !TAKES GPXRA OR MD AND PUTS THE EXPONENT (SIGN EXTENDED) AND
        !THE MANTISSA IN SUCCESSIVE GP REGISTERS.
        BEGIN
        GPXRB<=MD<7:0> NEXT
        RB←RB+1 NEXT
        GPXRB<31:8>←MD<31:8> NEXT
        GPXRB<7:0>←0
        END,

LOOP8:=
```

```
        BEGIN
        REPEAT
                BEGIN
                DECODE GPXRB<31:29>=>
                BEGIN
                [0,7]:= (GPXRB←GPXRB SL0 2 NEXT
                        IF SUMM1 XOR W1<7> =>(OVFF←1 NEXT LEAVE LOOP8) NEXT
! HERE SUMM1 IS THE EXTRA HARDWARE BIT TO THE LEFT OF THE EXPONENT
! AND IS USED TO CALCULATE OVF8 (=SUMM1 XOR THE LEFT MOST REAL BIT OF
! EXPONENT.)

                        SUMM1@W1<7:0>←W1<7>@W1<7:0>-2),
! THE EXPONENT PART OF W1 IS ALL THAT IS USED IN THE CONTINUATION.
                [1,6]:= (IF SUMM1 XOR W1<7> =>(OVFF←1 NEXT LEAVE LOOP8) NEXT
                        SUMM1@W1<7:0>←W1<7>@W1<7:0>-1 NEXT
                        GPXRB←GPXRB SL0 1 NEXT
                        IF SUMM1 XOR W1<7> =>(OVFF←1 NEXT LEAVE LOOP8) NEXT
                        GPXRB<7:0>←W1<7:0> NEXT
                        LEAVE LOOP8),
                2:5:=   (IF SUMM1 XOR W1<7> =>(OVFF←1 NEXT LEAVE LOOP8) NEXT
                        GPXRB<7:0>←W1<7:0> NEXT
                        LEAVE LOOP8)
                END
                END
        END,


PKF:=   !TAKES EXPONENT(GPXRB OR MD) AND MANTISSA(GPXRA OR MEM[MA])
        !AND PUTS THEM TOGETHER IN GPXRB AS ONE FLOATING POINT NUMBER.
        ! THIS NEEDS TO BE CHECKED AGAIN,
        BEGIN
        DECODE AMODE=>
                BEGIN
                0:= (W1←GPXRB NEXT    !EXPONENT
                    GPXRB←GPXRA),     !MANTISSA
        OTHERWISE:= (MA←MA+1 NEXT
                    GPXRB←MEM[MA] NEXT     !MANTISSA
                    W1←MD)                 !EXPONENT
                END NEXT
        IF GPXRB EQL 0=> (GPXRB←"80 NEXT LEAVE PKF) NEXT
        LOOP8()
        END,



LAND:=
        BEGIN
        GPXRB←MD AND GPXRB
        END,

LXOR:=
        BEGIN
        GPXRB←MD XOR GPXRB
```

```
        END,

IOR:=
        BEGIN
        GPXRB←MD OR GPXRB
        END,

ANI:=
        BEGIN
        GPXRB←NOT MD AND GPXRB
        END,

LOOP10:=
        BEGIN
        REPEAT
                BEGIN
                MD←MD+2 NEXT
                DECODE GPXRB<29:27>=>
                        BEGIN
                        [0,7]:= GPXRB←GPXRB SL0 2,
                    OTHERWISE:= (OVFF←1 NEXT
                                 GPXRB←GPXRB SL0 2)
                        END NEXT
                IF MD<7:0> EQL 0=>LEAVE LOOP10
                END
        END,


LOOP11:=
        BEGIN
                 MD←MD+2 NEXT
                IF GPXRB EQL 0=> LEAVE LOOP11 NEXT
                DECODE GPXRB<31:29>=>
                        BEGIN
                            [0,7]:= GPXRB←GPXRB SL0 2,
                       OTHERWISE:= (OVFF←1 NEXT
                                    GPXRB←GPXRB SL0 2)
                        END NEXT
                IF MD<7:0> NEQ 0=>LOOP10()
        END,



LOOP9:=
        BEGIN
        REPEAT
                BEGIN
                MD←MD-2 NEXT
                GPXRB<=GPXRB<31:2> NEXT
                IF MD<7:0> EQL 0=>LEAVE LOOP9
                END
        END,
```

```
ARS:=   !SHIFTS GPXRB THE NUMBER AND DIRECTION OF THE SIGNED EXPONENT
        !PART OF GPXRA OR MD. RIGHT SHIFT CAUSES SIGN-EXTENSION
        BEGIN
        IF MD<7:0> EQL 0=> LEAVE ARS NEXT
        DECODE MD<7>@MD<0>=>
                BEGIN
                0:= LOOP9(),
                1:= (MD←MD-1 NEXT
                    GPXRB<=GPXRB<31:1> NEXT
                    IF MD<7:0> NEQ 0=>LOOP9()),
                2:= LOOP11(),
                3:= (MD←MD+1 NEXT
                    IF GPXRB EQL 0=>LEAVE ARS NEXT
                    DECODE GPXRB<31:29>=>
                        BEGIN
                        [0,1,6,7]:= GPXRB←GPXRB SL0 1,
                        OTHERWISE:= (OVFF←1 NEXT
                                     GPXRB←GPXRB SL0 1)
                        END NEXT
                    IF MD<7:0> NEQ 0=>LOOP11())
                END
        END,

LOOP100:=
        BEGIN
        REPEAT
                BEGIN
                MD←MD+2 NEXT
                DECODE GPXRB<29:27>=>
                        BEGIN
                        [0,7]:= (GPXRB←GPXRB<29:0>@EX<31:30> NEXT
                                 EX←EX SL0 2),
                    OTHERWISE:= (OVFF←1 NEXT
                                 GPXRB←GPXRB<29:0>@EX<31:30> NEXT
                                 EX←EX SL0 2)
                        END NEXT
                IF MD<7:0> EQL 0=>LEAVE LOOP100
                END
        END,

LOOP110:=
        BEGIN
                MD←MD+2 NEXT
                DECODE GPXRB<31:29>=>
                        BEGIN
                        [0,7]:= (GPXRB←GPXRB<29:0>@EX<31:30> NEXT
                                 EX←EX SL0 2),
                    OTHERWISE:= (OVFF←1 NEXT
                                 GPXRB←GPXRB<29:0>@EX<31:30> NEXT
                                 EX←EX SL0 2)
                        END NEXT
```

```
                IF MD<7:0> NEQ 0=>LOOP100()
        END,


LOOP90:=
        BEGIN
        REPEAT
                BEGIN
                MD←MD-2 NEXT
                GPXRB<=GPXRB<31:2> NEXT
                EX←GPXRB<1:0>@EX<31:2> NEXT
                IF MD<7:0> EQL 0=>LEAVE LOOP90
                END
        END,


ARL:=   !SHIFTS GPXRB@EX THE NUMBER AND DIRECTION OF THE SIGNED EXPONENT
        !PART OF GPXRA OR MD.
        BEGIN
        EX←EX SL0 1 NEXT
        IF MD<7:0> EQL 0=> (EX←EX<31:1> NEXT LEAVE ARL) NEXT
        DECODE MD<7>@MD<0>=>
                BEGIN
                0:= LOOP90(),
                1:= (MD←MD-1 NEXT
                    EX←GPXRB<0>@GPXRB<31:1> NEXT
                    GPXRB<=GPXRB<31:1> NEXT
                    IF MD<7:0> NEQ 0=>LOOP90()),
                2:= LOOP110(),
                3:= (MD←MD+1 NEXT
                    IF GPXRB EQL 0=>(GPXRB←GPXRB<30:0>@EX<31> NEXT
                                     EX←EX SL0 1 NEXT
                                     (IF MD NEQ 0=>LOOP110() NEXT
                                                   LEAVE ARL) NEXT
                                     EX←EX SR0 1 NEXT
                                     LEAVE ARL) NEXT


                    DECODE GPXRB<31:29>=>
                        BEGIN
                        [0,1,6,7]:= (GPXRB←GPXRB<30:0>@EX<31> NEXT
                                     EX←EX SL0 1),
                        OTHERWISE:= (OVFF←1 NEXT
                                     GPXRB←GPXRB<30:0>@EX<31> NEXT
                                     EX←EX SL0 1)
                        END NEXT
                    IF MD<7:0> NEQ 0=>LOOP110())
                END
        END,


LOOP12:=
        BEGIN
```

```
        REPEAT
                BEGIN
                MD←MD-2 NEXT
                GPXRB←GPXRB SRR 2 NEXT
                IF MD<7:0> EQL 0=> LEAVE LOOP12
                END
        END,

LOOP13:=
        BEGIN
        REPEAT
                BEGIN
                MD←MD+2 NEXT
                GPXRB←GPXRB SLR 2 NEXT
                IF MD<7:0> EQL 0=> LEAVE LOOP13
                END
        END,

RRS:=           !ROTATES GPXRB NUMBER OF PLACES AND DIRECTIONS GIVEN
                !BY THE EXPONENT PART OFGPXRA OR MD.
        BEGIN
        IF MD<7:0> EQL 0=> LEAVE RRS NEXT
        DECODE MD<7>@MD<0>=>
                BEGIN
                0:= LOOP12(),
                1:= (MD←MD-1 NEXT
                    GPXRB←GPXRB SRR 1 NEXT
                    IF MD<7:0> NEQ 0=>LOOP12()),
                2:= LOOP13(),
                3:= (MD←MD+1 NEXT
                    GPXRB←GPXRB SLR 1 NEXT
                    IF MD<7:0> NEQ 0=>LOOP13())
                END
        END,


LOOP14:=
        BEGIN
        REPEAT
                BEGIN
                MD←MD-2 NEXT
                SUM←GPXRB NEXT
                GPXRB←EX<1:0>@GPXRB<31:2> NEXT
                EX←SUM<1:0>@EX<31:2> NEXT
                IF MD<7:0> EQL 0=>LEAVE LOOP14
                END
        END,


LOOP15:=
        BEGIN
        REPEAT
                BEGIN
```

```
                MD←MD+2 NEXT
                SUM←GPXRB NEXT
                GPXRB←GPXRB<29:0>@EX<31:30> NEXT
                EX←EX<29:0>@SUM<31:30> NEXT
                IF MD<7:0> EQL 0=>LEAVE LOOP15
                END
        END,


RRL:=    !ROTATES GPXRB@EX THE NUMBER OF PLACES AND DIRECTION OF THE SIGNED
         !EXPONENT PART OF GPXRA OR MD.
        BEGIN
        IF MD<7:0> EQL 0=> LEAVE RRL NEXT
        DECODE MD<7>@MD<0>=>
                BEGIN
                0:= LOOP14(),
                1:= (MD←MD-1 NEXT
                    SUM←GPXRB NEXT
                    GPXRB←EX<0>@GPXRB<31:1> NEXT
                    EX←SUM<0>@EX<31:1> NEXT
                    IF MD<7:0> NEQ 0=>LOOP14()),
                2:= LOOP15(),
                3:= (MD←MD+1 NEXT
                    SUM←GPXRB NEXT
                    GPXRB←GPXRB<30:0>@EX<31> NEXT
                    EX←EX<30:0>@SUM<31> NEXT
                    IF MD<7:0> NEQ 0=>LOOP15())
                END
        END,


LOOP16:=
        BEGIN
        REPEAT
                BEGIN
                MD←MD-2 NEXT
                GPXRB←GPXRB SR0 2 NEXT
                IF MD<7:0> EQL 0=>LEAVE LOOP16
                END
        END,

LOOP17:=
        BEGIN
        REPEAT
                BEGIN
                MD←MD+2 NEXT
                GPXRB←GPXRB SL0 2 NEXT
                IF MD<7:0> EQL 0=>LEAVE LOOP17
                END
        END,
```

```
LRS:=  !ZEROS ARE SHIFTED IN.
       BEGIN
       IF NOT AMODE=>MD←GPXRA NEXT
       IF MD<7:0> EQL 0=>LEAVE LRS NEXT
       DECODE MD<7>@MD<0>=>
              BEGIN
              0:= LOOP16(),
              1:= (MD←MD-1 NEXT
                  GPXRB←GPXRB SR0 1 NEXT
                  IF MD<7:0> NEQ 0=>LOOP16()),
              2:= LOOP17(),
              3:= (MD←MD+1 NEXT
                  GPXRB←GPXRB SL0 1 NEXT
                  IF MD<7:0> NEQ 0=>LOOP16())
              END
       END.


LOOP18:=
       BEGIN
       REPEAT
              BEGIN
              MD←MD-2 NEXT
              EX←GPXRB<1:0>@EX<31:2> NEXT
              GPXRB←GPXRB SR0 2 NEXT
              IF  MD<7:0> EQL 0=> LEAVE LOOP18
              END
       END.

LOOP19:=
       BEGIN
       REPEAT
              BEGIN
              MD←MD+2 NEXT
              GPXRB←GPXRB<29:0>@EX<31:30> NEXT
              EX←EX SL0 2 NEXT
              IF MD<7:0> EQL 0=>LEAVE LOOP19
              END
       END.


LRL:=  !ZERO IS SHIFTED IN.
       BEGIN
       IF MD<7:0> EQL 0=>LEAVE LRL NEXT
       DECODE MD<7>@MD<0>=>
              BEGIN
              0:= LOOP18(),
              1:= (MD←MD-1 NEXT
                  EX←GPXRB<0>@EX<31:1> NEXT
                  GPXRB←GPXRB SR0 1 NEXT
                  IF MD<7:0> NEQ 0=>LOOP18()),
              2:= LOOP19(),
              3:= (MD←MD+1 NEXT
```

```
                GPXRB←GPXRB<30:0>@EX<31> NEXT
                EX←EX SL0 1 NEXT
                IF MD<7:0> NEQ 0=>LOOP19())
            END
        END,

DSI:=
        BEGIN
        DISINT←1
        END,

ENI:=
        BEGIN
        DISINT←0
        END,

RFI:=           !return from interrupt
        BEGIN
        PC←MEM[MA]<15:0> NEXT
        EXMODE←MEM[MA]<24> NEXT
        DISINT←MEM[MA]<23> NEXT
        DIVFF←MEM[MA]<22> NEXT
        OVFF←MEM[MA]<21> NEXT
        ILLOPC←MEM[MA]<20> NEXT
        CRYFF←MEM[MA]<19> NEXT
        INTPRIOR←MEM[MA]<18:16>
        END,

RET:=           !return from subroutine
        BEGIN
        PC←MEM[MA]<15:0> NEXT
        DISINT←MEM[MA]<23> NEXT
        DIVFF←MEM[MA]<22> NEXT
        OVFF←MEM[MA]<21> NEXT
        ILLOPC←MEM[MA]<20> NEXT
        CRYFF←MEM[MA]<19>
        END,

XEC:=
        BEGIN
        DECODE AMODE=>
                BEGIN
                0:= INR←GPXRA,
        OTHERWISE:= INR←MEM[MA]
                END
        NEXT
        MA<=INR<15:0> NEXT
        IF INRPT=>(PC←PC-1 NEXT LEAVE XEC) NEXT
        ADDRESS() NEXT
        OPERAND() NEXT
        OPERATION() NEXT
```

```
        GPXR[RA]←GPXRA NEXT
        GPXR[RB]←GPXRB NEXT
        INRPT←OVFF OR DIVFF
        END


**Operation.decode**


OPERATION:=    !opcode decode.
BEGIN

DECODE OPCODE=>
        BEGIN

        !LOAD(FETCH)/STORE

        "00:=LDR(),             !LOAD REGISTER
        "01:=LDE(),             !LOAD EXTENSION REGISTER
!       "30:=LW0(),             !LOAD WORKING REGISTER 0
!       "31:=LW1(),             !LOAD WORKING REGISTER 1
!       "32:=LW2(),             !LOAD WORKING REGISTER 2
!       "33:=LW3(),             !LOAD WORKING REGISTER 3
        "02:=LDR2(),            !LOAD MULTIPLE
        "03:=LDR3(),
        "04:=LDR7(),
        "05:=LDN(),             !LOAD NEGATIVE
        "06:=LDNF(),            !LOAD NEGATIVE FLOATING
        "07:=LDA(),             !LOAD ABSOLUTE VALUE
        "08:=LDAF(),            !LOAD ABSOLUTE VALUE FLOATING
        "09:=LDC(),             !LOAD ONE'S COMPLEMENT
        "0A:=LAO(),             !LOAD ACTIVE ONLY
        "0B:=LMO(),             !LOAD MONITOR ONLY

        "40:=STR(),             !STORE REGISTER
        "41:=STE(),             !STORE EXTENSION
!       "65:=SW0(),             !STORE WORKING REGISTER 0
!       "66:=SW1(),             !STORE WORKING REGISTER 1
!       "67:=SW2(),             !STORE WORKING REGISTER 2
!       "68:=SW3(),             !STORE WORKING REGISTER 3
        "42:=STD(),             !STORE DOUBLE
        "47:=STZ(),             !STORE ZERO
        "43:=SZD(),             !STORE ZERO DOUBLE
        "62:=STR2(),            !STORE MULTIPLE SINGLE
        "63:=STR3(),
        "44:=STD2(),            !STORE DOUBLE MULTIPLE
        "45:=STD3(),
        "46:=STD7(),
        "49:=STH(),             !STORE TO HARD ADDRESS
        "48:=SPS(),             !STORE PC AND STATUS SINGLE
        "4A:=SPC(),             !STORE PC AND STATUS DOUBLE
!       "4B:=SBPA1(),           !STORE BAD ADDRESS PARITY ONES
!       "4C:=SBPA0(),           !STORE BAD ADDRESS PARITY ZEROS
```

```
!       "4D:=SBPD1(),           !STORE BAD DATA PARITY ONES
!       "4E:=SBPD0(),           !STORE BAD DATA PARITY ZEROS

!JUMP

        "50:=JMP(),             !JUMP
        "5E:=JSB(),             !JUMP SUBROUTINE
        "4F:=JPZ(),             !JUMP IF POSITIVE OR ZERO
        "51:=JMI(),             !JUMP IF NEGATIVE
        "52:=JZE(),             !JUMP IF ZERO
        "53:=JZEF(),            !JUMP IF ZERO FLOATING
        "54:=JNZ(),             !JUMP IF NON-ZERO
        "55:=JNZF(),            !JUMP IF NON-ZERO FLOATING
        "56:=JPS(),             !JUMP IF POSITIVE AND NON-ZERO(?)
        "57:=JPSF(),            !JUMP IF POSITIVE AND NON-ZERO FLOATING(?)
        "58:=JMZ(),             !JUMP IF NEGATIVE OR ZERO
        "59:=JMZF(),            !JUMP IF NEGATIVE OR ZERO FLOATING
        "5A:=JDN(),             !DECREMENT RB. JUMP IF NON-ZERO
        "5C:=JOS(),             !JUMP IF OVERFLOW SET. RESET OVERFLOW
        "5D:=JCS(),             !JUMP IF CARRYOUT SET. RESET CARRYOUT

!TEST AND SKIP

        "29:=OISN(),            !OR INVERTED AND SKIP IF NOT ONES
        "2D:=OISO(),            !OR INVERTED AND SKIP IF ONES
        "2A:=ASNZ(),            !AND AND SKIP IF NOT ZEROS
        "2C:=ASZ(),             !AND AND SKIP IF ZEROS
!       "34:=CSNE(),            !COMPARE AND SKIP IF NOT EQUAL
!       "35:=CSEQ(),            !COMPARE AND SKIP IF EQUAL

!INTEGER ARITHMETIC

        "19:=ADD(),             !INTEGER ADD
        "1A:=SUB(),             !INTEGER SUBTRACT
        "1B:=MPY(),             !INTEGER MULTIPLY
        "1C:=DIVI(),            !SHORT DIVIDEND DIVIDE
        "1D:=LDV(),             !LONG DIVIDEND DIVIDE
        "2B:=ACO(),             !ADD CARRYOUT
        "1E:=CFL(),             !CONVERT INTEGER TO FLOATING

!FLOATING POINT ARITHMETIC

        "0C:=ADDF(),            !ADD FLOATING
        "0D:=SUBF(),            !SUBTRACT FLOATING
        "0E:=MPYF(),            !MULTIPLY FLOATING
        "0F:=DIVF(),            !DIVIDE FLOATING
        "10:=SRTF(),            !SQUARE ROOT FLOATING
        "11:=VADDF(),           !VECTOR ADD FLOATING
!       "12:=VSUBF(),           !VECTOR SUBTRACT FLOATING
        "13:=VMPYF(),           !VECTOR MULTIPLY FLOATING
        "14:=VIPF(),            !VECTOR INNER PRODUCT FLOATING
        "15:=VSMF(),            !VECTOR-SCALAR MULTIPLY FLOATING
        "16:=CFX(),             !CONVERT FLOATING TO INTEGER
```

```
        "17:=UPF(),             !UNPACK FLOATING
        "18:=PKF(),             !PACK FLOATING

!LOGICAL

        "1F:=LAND(),            !LOGICAL AND (Name changed because of conflict
                                !with ISPS)
        "28:=LXOR(),            !EXCLUSIVE OR (ditto)
        "21:=IOR(),             !INCLUSIVE OR
        "22:=ANI(),             !AND INVERTED

!SHIFT/ROTATE

        "23:=ARS(),             !ARITHMETIC SHORT SHIFT
        "24:=ARL(),             !ARITHMETIC LONG SHIFT
        "25:=RRS(),             !ROTATE SHORT
        "26:=RRL(),             !ROTATE LONG
        "27:=LRS(),             !LOGICAL SHORT SHIFT
        "28:=LRL(),             !LOGICAL LONG SHIFT

!MISCELLANEOUS

        "58:=DSI(),             !DISABLE INTERRUPTS
        "5F:=ENI(),             !ENABLE INTERRUPTS
        "60:=RFI(),             !RETURN FROM INTERRUPT
        "64:=RET(),             !RETURN FROM SUBROUTINE
        "61:=XEC()              !EXECUTE

        END
END
```

```
**Execution.cycle**


CYCLE !MAIN! :=
        BEGIN
        DELAY(1) NEXT
        PC←1 NEXT
        REPEAT
                BEGIN
                INSTRUCTION() NEXT
                ADDRESS() NEXT
                OPERAND() NEXT
                OPERATION() NEXT
                GPXR[RA]←GPXRA NEXT
```

```
                GPXR[RB]←GPXRB NEXT
                INRPT←OVFF OR DIVFF
        !       IF INRPT=>(...
                END
        END
)

STOP
```

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.